

PG 524

**RAPTOR - Room Adaptive
Precisely Topologically
Orienting Robot**

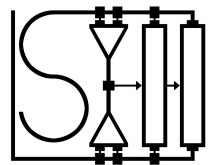
Zwischenbericht

Wintersemester 2007/2008

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl XII (Embedded System Design)
Fakultät für Informatik
Technische Universität Dortmund

Betreuer:
Olivera Jovanovic
Robert Pyka





Zwischenbericht

Projektgruppe 524

Room Adaptive Precisely Topologically Orienting Robot – RAPTOR

Wintersemester 2007/2008

Teilnehmer:

Ender Ayalp, Sariette Bille, Omar Bousbiba, Daniel Etges, Igor Ionov,
Selma Jabour, Oliver Jokiel, Dominik Kopczynski, Markus Künne, Yi Lin,
Samir Rachidi, Daniel Richter

Betreuer:

Olivera Jovanovic, Robert Pyka

Technische Universität Dortmund

Fachbereich Informatik

LSXII

Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

1	Einleitung	7
1.1	Motivation	7
1.2	Ziel	8
1.2.1	Minimalziele	9
1.2.2	Erweiterungsmöglichkeiten	9
1.3	Vorgehensweise	10
1.3.1	Hardwaregruppe	10
1.3.2	Softwaregruppe	11
2	Grundlagen	13
2.1	Hardware	13
2.1.1	Linux Board	13
2.1.2	Ultraschallsensoren	18
2.1.3	Kontaktsensoren	20
2.1.4	Lichtsensoren	23
2.1.5	Maussensor	25
2.1.6	Kompasssensor	27
2.1.7	Getriebemotoren	29
2.1.8	Mikrocontroller	31
2.1.9	Kamera	32
2.2	Software	33
2.2.1	Buildroot	34
2.2.2	Player-Stage	34
2.2.3	Application Programming Interface	35
2.2.4	Umgebungsmodell	36
2.2.5	Mikrocontroller	36
3	Hardware	39
3.1	Testumgebung	39
3.1.1	Steckbrett	39
3.1.2	BreakOutBox	41
3.1.3	Motortreiber	47
3.2	ATMega	48
3.2.1	Programmierhardware	51
3.2.2	Inbetriebnahme des ATMega32	53

3.2.3	Programmierung in Basic	55
3.2.4	Einbindung eines Quarzes	56
3.2.5	Umstieg auf C	59
3.2.6	Interrupts	62
3.2.7	I2C auf dem Atmel	64
3.3	Herstellung von Platinen	65
3.3.1	Layout	65
3.3.2	Belichtung	66
3.3.3	Entwicklung	66
3.3.4	Ätzen	66
3.3.5	Säubern und Versiegeln	67
3.3.6	Bohren	67
3.3.7	Löten	67
3.4	Hauptplatine	67
3.4.1	ATMega32	67
3.4.2	Quarz	69
3.4.3	Kompasssensor und Gumstix	70
3.4.4	Reset	70
3.4.5	Radencoder	70
3.4.6	Ultraschallsensoren	71
3.4.7	Leuchtdioden	72
3.4.8	Motoren	72
3.4.9	Akkumulator	74
3.4.10	Kontaktsensoren	75
3.4.11	Abgrundsensoren	75
3.4.12	Maussensor	76
3.5	Karosserie	77
3.5.1	Die untere Ebene	78
3.5.2	Die obere Ebene	81
4	Software	85
4.1	Gumstix	85
4.1.1	Aufbau des Flash-Speichers	85
4.1.2	U-Boot	86
4.1.3	Inbetriebnahme	86
4.1.4	Konfiguration des Betriebssystems	86
4.1.5	Flashen des Kernels/Dateisystems	87
4.1.6	Eigene Programme	89
4.1.7	Zusätzliche Hardware	90
4.1.8	I2C auf dem Gumstix	91
4.1.9	I2C Kommunikation	93
4.1.10	Buildroot	94
4.2	Gitterkarte	95

4.2.1	GridMap	96
4.2.2	GridMapData	102
4.2.3	Feld	106
4.3	Voronoi	109
4.3.1	Einführung in das Voronoi-Diagramm	109
4.3.2	Definition des Voronoi-Diagramms	110
4.3.3	Algorithmus zur Berechnung des Voronoi-Diagramms	113
4.4	Player-Stage	116
4.4.1	Installation	116
4.4.2	Konfiguration des Roboters	119
4.4.3	Simulation starten	120
4.4.4	Eigene Programme erstellen	120
4.4.5	Makefile	123
4.5	Application Programming Interface	123
4.6	ATMega32	124
4.6.1	Maussensor	125
4.6.2	Motoren	125
4.6.3	Inter-Integrated Circuit	129
5	Ausblick	137
A	Hardwarekomponenten	139
A.1	Skizzen der Karosserie	139
A.2	Schaltpläne	143
A.3	Platinenlayouts	148
B	Quelltexte	151
B.1	Beispielprogramme für die BreakoutBox	151
B.2	Basic-Beispielprogramme für den ATMega32	153
B.3	C-Beispielprogramme für den ATMega32	156
B.4	Motortreiber	168
B.5	Motortreiber der Testschaltung	171
	Literaturverzeichnis	181

1 Einleitung

Die Projektgruppe 524, auch RAPTOR (Room Adaptive Precisely Topologically Orienting Robot) genannt, entwickelt mit dem c't Bot als Grundlage, einen Staubsaugerroboter ohne Saugmechanismus. Motiviert von den Unzulänglichkeiten bestehender Roboter strebt die Projektgruppe einige Verbesserungen, siehe Abschnitt 1.1 auf dieser Seite, an. Auf das Ziel und die Vorgehensweise wird in den Abschnitten 1.2 und 1.3 näher eingegangen.

1.1 Motivation

Schon immer war es ein Traum der Menschen künstliche Wesen zu erzeugen. Diese sollten ihnen mühsame, beziehungsweise gefährliche Arbeiten abnehmen oder zur Unterhaltung der Menschen dienen. Inzwischen gibt es eine Vielzahl von Robotern für die unterschiedlichsten Einsatzbereiche. Die bekanntesten Roboter sind die Industrieroboter. Diese übernehmen unterschiedliche Aufgaben, wie beispielsweise schweißen und montieren, in Anwendungsbereichen wie zum Beispiel Schiffsbau und Elektrotechnik. In der Forschung und Unterhaltungsindustrie werden humanoide Roboter entwickelt, dessen Herausforderung vor allem darin liegt, dem Roboter das Gehen auf zwei Beinen beizubringen. Desweiteren gibt es unterschiedliche mobile Roboter, die für den Einsatz im Weltraum oder als Überwachungs- und Serviceroboter dienen. In der Computer-Fachzeitschrift c't erschien 2006 das Projekt c't-Bot [7]. In diesem Projekt wurde ein Roboter vorgestellt, der sich einfach nachbauen lässt. Mit Hilfe des Simulators c't sim lässt sich sein Verhalten virtuell testen. Er besitzt die Fähigkeit seinen Standort zu bestimmen und erstellt eine Karte seiner Umgebung. Mit seinem Maussensor wertet er seine Ausrichtung aus. Durch seinen Radencoder lässt sich die zurückgelegte Entfernung bestimmen. Mit Hilfe seiner Infrarotsensoren erkennt er Hindernisse und erstellt eine metrische Rasterkarte seiner Umgebung. Zusätzlich besitzt der Roboter eine Abgründetektion. Der Funktionsumfang des Projektes c't-Bot umfasst das eigenständige Erkunden der Umgebung, Slalom fahren und das Einsammeln von Golfbällen, die an eine bestimmte Position gebracht werden. Mit der entsprechenden Programmierung ist der Roboter allerdings auch in der Lage komplexere Tätigkeiten auszuführen.

Zu den beliebtesten Serviceroboter gehören die Staubsaugerroboter. Je nach Ausstattung und Saugleistung gibt es sie in unterschiedlichen Preisklassen zu kaufen. Die Roboter werden immer leistungsfähiger und mittlerweile stattdessen die Hersteller diese auch mit einem Treppensensorsystem aus, das verhindert, dass der Roboter Treppen herunterfällt. Weiterhin fahren die meisten Roboter ihre Ladestation eigenständig an, wenn der Akku leer ist. Einige Firmen bieten ferner eine Absaugstation zur Entleerung des Filters an. Zur Hinderniserkennung dienen entweder Drucksensoren am Gehäuse oder Ultraschallsensoren.

Ein Ersatz für herkömmliche Staubsauger sind diese Roboter noch nicht. Deshalb gibt es einige Ansätze, die heutigen Staubsaugerroboter um wichtige Eigenschaften zu erweitern:

- Die Analyse des Verschmutzungsgrades
- Das Erkennen von Menschen und Haustieren
- Das Säubern von Ecken und Rändern

Um die Erweiterungen zu realisieren, können die Mechanik, bestimmte Sensoren und Algorithmen vom c't-Bot übernommen werden. Für eine genügende Rechenleistung sollte ein Embedded Linux Board benutzt werden. Dieses ist eine vorgefertigte standardisierte Steuereinheit auf der ein Embedded Linux oder RTLinux zum Einsatz kommt.

1.2 Ziel

Die Projektgruppe soll in zwei Semestern einen autonomen, mobilen Roboter entwickeln, der als Staubsaugerroboter in einer Wohnung eingesetzt werden könnte. Allerdings braucht der Roboter keinen Saugmechanismus zu enthalten, da die heutige Technologie in derartig kleiner Bauform noch keine ausreichende Reinigungsleistung erreicht. Auf Grund dessen soll die mechanische Konstruktion auf das zur Fortbewegung Notwendige reduziert werden. Zugleich kann damit mehr Zeit in die Softwareentwicklung und Strategieplanung investiert werden.

Der Staubsaugerroboter RAPTOR hat folgende Besonderheiten:

- Der Verschmutzungsgrad des Bodens soll mit Hilfe eines Kameramoduls erkannt werden. Da größtenteils immer dieselben Stellen in einem Raum besonders verschmutzt sind, soll der Roboter immer direkt dorthin fahren. Mit Hilfe dieser Strategie kann so schnell ein gepflegtes Gesamtbild der Wohnung erreicht werden. Der Roboter soll einen Ausschnitt des Bodens fotografieren, anschließend staubsaugen und noch einmal fotografieren um dann durch eine Analyse der Bilder den Verschmutzungsgrad in diesem Abschnitt zu bestimmen. Da der RAPTOR keinen Saugmechanismus enthält, müssen die Teilnehmer der Projektgruppe das Staubsaugen mit anderen Hilfsmitteln übernehmen.
- Menschen und Haustiere sollen mittels Sensoren erkannt werden, damit diese rechtzeitig umfahren werden können und das Staubsaugen nicht als Störung empfunden wird.
- Der Roboter soll eine gründliche Reinigung der Ecken und Ränder von Möbel und Tischen in einem Raum gewährleisten. Dies setzt eine Software voraus, die in der Lage ist Ecken und Ränder zu erkennen. Ferner wird die Hardware benötigt, die in der Lage ist ein nahes Heranfahren an Hindernisse zu ermöglichen, damit auch dort der Schmutz entfernt werden kann.

Die mechanische Realisierung kann vom c't-Bot weitestgehend übernommen werden. Da der Durchmesser des c't-Bots nur 12cm beträgt, müssen seine Komponenten für den RAPTOR angemessen skaliert werden. Anstelle des Mikrocontrollers wird ein Embedded Linux Board benutzt um eine ausreichende Rechenleistung für die vorgestellten Erweiterungen sicherzustellen. Damit steht den Teilnehmern der Projektgruppe eine fertige Entwicklungsumgebung zur Verfügung. Um neben den Infrarotsensoren auch Ultraschallsensoren, Navigations- und Ortungssensoren anschließen zu können, besitzt das Board vielfältige I/O-Anschlüsse. Der Roboter sollte selbstständig, nach einer von der Projektgruppe ausgewählten Strategie, den Raum (beziehungsweise auch mehrere Räume wenn möglich) erkunden und eine Karte erstellen. Außerdem sollte er in der Lage sein sich selbst zu lokalisieren. Zusätzlich soll eine graphische Kontroll- und Steuerungsschnittstelle implementiert werden. Mit Hilfe dieser Schnittstelle werden Parameter, wie beispielsweise der Akkuladestand, die Sensorwerte und das aktuelle Kamerabild, des Roboters überwacht und die graphisch aufbereitete Darstellung des Raummodells zugänglich gemacht. Die Kommunikation mit dem Roboter erfolgt funkbasiert. Das ausgewählte Embedded Linux Board stellt eine WLAN-kompatible Schnittstelle zur Verfügung.

1.2.1 Minimalziele

Für den Erwerb des Leistungsnachweises sind die folgenden Minimalziele zwingend erforderlich:

- Hardwareentwurf des Roboters inklusive Auswahl der zu verwendenden Bauelemente. Beim Entwurf ist zu gewährleisten, dass sich der Roboter, unter Nutzung der Navigationsvorrichtung, zielsicher in einem Raum bewegen kann.
- Implementierung einer Low-Level Steuerungssoftware zur Bewegungssteuerung des Roboters und Positions- beziehungsweise Distanzmessung. Der Roboter soll damit vordefinierte Punkte im Raum anfahren können.
- Implementierung einer High-Level Steuerungssoftware zur Navigation. Dabei soll ein Modell des Raums zur Verfügung stehen und eine festgelegte Mission implementiert sein. Eine Mission könnte beispielsweise sein, den Raum möglichst vollständig abzufahren.
- Demonstration der Funktionsfähigkeit der erstellten Hard- und Software im praktischen Einsatz.

1.2.2 Erweiterungsmöglichkeiten

Der Wettbewerb *Patente Studierende* [20] prämiert, im September 2008 in Düsseldorf, Haushaltsroboter die dem Menschen sowohl behilflich, als auch individuell einsetzbar sind. Dabei sollte er nicht allein die Funktion eines Staubsaugers ausüben. Diese Tatsache

hat die Projektgruppe RAPTOR bereits animiert über Erweiterungsmöglichkeiten nachzudenken.

Der Anmeldezeitraum beginnt am 31.10.07 und endet am 29.02.2008. In der Anmeldung sollte kurz angegeben werden, welche Besonderheit der Haushaltsroboter haben wird, ohne dabei Lösungen und Eigenschaften zu beschreiben. Die Bewertungskriterien der Jury, die sich im Wesentlichen aus Vertretern von Unternehmen und Universitäten zusammensetzt, sind:

- Originalität in der Problemlösung
- Neuheit gegenüber dem Stand der Technik
- Anwendernutzen und Verwertbarkeit im Markt
- Effizienz (Praktischer Nutzen zum technischen Aufwand)
- Sorgfalt in der inhaltlichen und formalen Ausführung
- Präsentation

1.3 Vorgehensweise

Die Projektgruppe 524 findet im Wintersemester 2007/2008 und im Sommersemester 2008, mit zwölf studentischen Teilnehmern am Lehrstuhl 12 der Fakultät Informatik der Technischen Universität Dortmund, statt. Betreut wird das Projekt von den wissenschaftlichen Mitarbeitern Olivera Jovanovic und Robert Pyka des Lehrstuhls 12 für Technische Informatik und Eingebettete Systeme.

Unter der URL <http://ls12-www.cs.tu-dortmund.de/raptor> ist der Internetauftritt der Projektgruppe zu finden.

Für eine bessere Organisation verfolgen die Teilnehmer die Ziele der Projektgruppe in zwei Teams:

Hardwaregruppe Sariette Bille, Omar Bousbiba, Daniel Etges, Igor Ionov, Selma Jabour, Yi Lin und Samir Rachidi.

Softwaregruppe Ender Ayalp, Oliver Jokiel, Dominik Kopczynski, Markus Künne und Daniel Richter.

1.3.1 Hardwaregruppe

Kapitel 2.1 beschreibt die Auswahl der Hardwareelemente, die die Hardwaregruppe in den Anfangswochen der Projektgruppe traf. Nachdem die Grundlagen geklärt waren, konnte die Realisierung des Prototyps eines autonomen Staubsaugerroboters beginnen. Nachzulesen ist dies im Kapitel 3. Der abschließende Ausblick des Zwischenberichts, siehe Kapitel 5, beschreibt unter anderem das weitere Vorgehen der Hardwaregruppe für das Sommersemester 2008.

1.3.2 Softwaregruppe

Da sich die Projektgruppe bei der Herstellung ihres Staubsaugerroboters nah an dem Projekt des c't Bots orientiert, gibt es einiges an vorgefertigter Software. In den ersten Wochen des Wintersemesters 2007/2008 verschaffte sich die Softwaregruppe einen Überblick über vorhandene Software zu diesem Themengebiet. Das Kapitel 2.2 beschreibt die Vorüberlegungen. Die letztendliche Realisierung der Software wird in dem Kapitel 4 geklärt. Die ausbleibenden Programmierarbeiten der Softwaregruppe werden im Ausblick (Kapitel 5) festgehalten.

2 Grundlagen

Dieses Kapitel beschreibt die Grundlagen der Hard- und Software. Dazu gehören unter anderem die Auswahl der verschiedenen Bauelemente, sowie die jeweilige Hardwarespezifikation. Ferner wird beschrieben, wie auf existierende Software zurückgegriffen wurde und welche Software selbsterstellt wurde. Dieses Kapitel beschreibt demnach nur die Auswahl der Hard- und Software. Die Realisierung dieser wird im Kapitel 3 bzw. 4 behandelt.

2.1 Hardware

Zu Beginn der Projektgruppe wurde die entsprechende Hardware für den autonomen Roboter ausgesucht. Dazu gehört die Auswahl des passenden Linux Board, der Ultraschallsensoren, der Kontaktsensoren, der Lichtsensoren, des Maussensors, des Kompassensors, der Getriebemotoren, des Mikrocontrollers und der Kamera. Die folgenden Abschnitte befassen sich mit diesem Thema.

2.1.1 Linux Board

Der Staubsaugerroboter soll in der fertigen Version eine hohe Anzahl an teilweise sehr komplexen Funktionalitäten, wie z.B. Staubererkennung, Navigation, Objekterkennung etc., ausführen können. Die benötigten Daten aus der Umgebung werden durch die Sensoren bereitgestellt, anhand derer der Roboter dann seine nächsten Aktionen planen kann. Eine Bearbeitung von Sensordaten erfordert ein hohes Maß an Rechenleistung, da Methoden aus den Bereichen der Bild- und Signalverarbeitung benötigt werden. Zudem soll der Roboter einen Großteil der Berechnungen in Echtzeit durchführen können, da er z.B. möglichst rechtzeitig Objekte erkennen und ggf. ausweichen soll. Hierfür ist neben einer effektiven Programmierung auch leistungsfähige Hardware erforderlich. Außerdem muss eine Vielzahl an verschiedensten Eingabe- und Ausgabegeräten, wie z.B. Sensoren, Kameras usw., unterstützt werden können.

Im Gegenzug zur benötigten Rechenleistung und Flexibilität stehen jedoch diverse Einschränkungen, die beim Roboter berücksichtigt werden müssen. Diese bestehen vor allem in der begrenzten Platz- und Energiekapazität, welche nicht jede beliebige Hardware, wie sie bei normalen Desktop-PCs möglich wäre, zulässt.

Aus Sicht der Software wird die Programmierung des Roboters extern auf Desktop-PCs durchgeführt, da es sehr aufwendig ist, direkt auf dem Roboter zu programmieren. Außerdem könnten dann mehrere Personen für verschiedene Softwarebereiche nicht mehr gleichzeitig an der Programmierung arbeiten. Der Programmcode wird dann auf dem Roboter

ausgelagert und lokal auf dem Gerät ausgeführt. Dies ist jedoch ohne ein Betriebssystem, welches die passende Programmierumgebung unterstützt, nicht ohne weiteres möglich. Ein geeignetes Betriebssystem für den Roboter und eingebettete Systeme allgemein wäre Linux: Es ist frei verfügbar, beansprucht wenig Speicherplatz, läuft bei korrekter Konfiguration sehr stabil, erlaubt viele Freiheiten in der Treiberprogrammierung für verschiedene Peripherie und stellt viele, bereits vorgefertigte Tools und Treiber zur Verfügung.

Als ideale Rechnerhardware für eingebettete Systeme, zu denen der Roboter sicherlich zählt, erweisen sich die sog. Gumstix-Boards der amerikanischen Firma gumstix inc. Diese zeichnen sich durch eine hohe Rechenleistung bei kompakter Größe (die einheitliche Größe der Rechner-Boards beträgt 80mm x 20mm, während die Größe der Erweiterungs-Boards minimal abweichen kann), und geringer Betriebsspannung (3.3 V - 5 V) aus. Um eine Vielzahl unterschiedlicher Peripherie zu unterstützen, können die Rechner-Boards mit einer breiten Palette an verschiedenen Erweiterungs-Boards kombiniert werden. Insgesamt gibt es 3 verschiedene Gruppen von gumstix-Rechner-Boards (basix, connex und verdex), diese unterscheiden sich hauptsächlich in der Rechenleistung und der Anzahl an unterstützten Erweiterungs-Boards.

Rechnerboard: gumstix verdex XL6P

Die Projektgruppe entschied sich für das Modell **gumstix verdex XL6P**, welches bis zu 2 Erweiterungs-Boards unterstützt und über die höchstmögliche Rechenleistung aller bestellbaren Rechner-Boards verfügt. Zwar ist das gumstix verdex XL6P das teuerste, verfügbare Board (um 70 Dollar teurer im Vergleich zum billigsten Modell der Produktgruppe basix), dafür ist es in der Lage, allen von Robotern gestellten Anforderungen gerecht zu werden. Wichtig ist insbesondere die Möglichkeit, bis zu 2 Erweiterungs-Boards an das gumstix verdex XL6 anzuschliessen, wodurch die Anzahl möglicher Ein-/Ausgabegeräte deutlich erhöht wird. Zudem gibt es einige spezielle Erweiterungs-Boards (u.a. das console-vx und das netwifimicroSD), welche nur mit Rechner-Boards der verdex-Klasse kompatibel sind.

 **verdex XL6P**



Abbildung 2.1: gumstix verdex XL6P [11]

Technische Details:

- Prozessor: Marvell PXA270 XScale-Prozessor mit 600 MHz
- Speicher: 128MB RAM, 32MB Flash-Speicher
- Features: USB-host Signale, CCD-camera Signale
- Anschlüsse für Erweiterungs-Boards:
 - 60-pin Hirose connector
 - 120-pin MOLEX connector
 - 24-pin flex ribbon
- Größe: 80mm x 20mm
- Gewicht: 8g

Das Board enthält bei Bestellung bereits einen vorgefertigten Linux-Kernel 2.6 auf dem Flash-Speicher, welcher nach Belieben verändert oder neu beschrieben werden kann. Dieses gilt jedoch nicht für den Bootloader. Sollte dieser beschädigt oder fehlerhaft verstellt werden, so muss das Produkt zurück zum Hersteller gesendet werden, damit der Speicher (gegen Aufpreis) neu geflasht werden kann.

Mit diesem Board ist die benötigte Rechenleistung inkl. Speicherversorgung gewährleistet. Nun muss sichergestellt werden, dass eine ausreichende Menge an Anschlüssen für die verschiedenen Ein-/Ausgabegeräte vorhanden ist. Dies wird im Nachfolgenden durch die 2 Erweiterungs-Boards erreicht.

Erweiterungs-Board: Console-vx

Das Erweiterungs-Board console-vx wird an den 60-pin Hirose connector des gumstix verdex XL6P angeschlossen.

Technische Details:

- 3 x RS-232 Ports mit miniDIN8 Verbindungssteckplätzen
- USB mini-B Anschluß with USB-Host Signalen
- 18-bit LCD Anschluß
- Serielle Funktionsports (alternative GPIO Verbindungen) auf 0.100 Zoll kleinen Löchern
- Verbindungsanschluß : 60-pin I/O header
- Größe: 80mm x 34mm
- Betriebsspannung: 3.5V - 5V

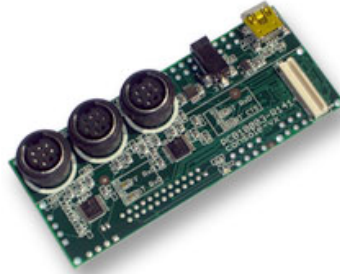
■ console-vx

Abbildung 2.2: console-vx [11]

Ursprünglich wurde es bestellt, damit die PG-Teilnehmer erste Erfahrungen mit dem Gumstix sammeln können. Hierfür wurden 3 Pins eines RS-232 Ports verkabelt, mit einer Stromquelle verbunden und mit Hilfe eines seriellen Anschlusses an den Hauptrechner im Hardware-Labor angeschlossen. Bei ausreichender Spannung wird das Gumstix automatisch vom Rechner erkannt und kann nun mit Hilfe des Programms Minicom konfiguriert werden. Die ersten Versuche, Daten vom Hauptrechner auf das Gumstix zu flashen, verliefen problemlos. Jedoch erweist sich die Geschwindigkeit des seriellen Anschlusses als sehr langsam, weshalb im Laufe der Zeit nach anderen, schnelleren Möglichkeiten zur Datenübertragung auf das Gumstix gesucht wurden (siehe nächstes Kapitel). Dieses Erweiterungs-Board verfügt zusätzlich über GPIO-Schnittstellen, welche den Anschluß weiterer, einfacher Peripherie wie z.B. Sensoren unterstützen. Weiterhin erwähnenswert ist die Unterstützung von USB-Host Signalen, die dem console-vx auch erlaubt, selber als Host z.B. für Netzwerke zu agieren. Diese Möglichkeit wurde von der Projektgruppe allerdings noch nicht in Anspruch genommen.

Erweiterungs-Board: netwifimicroSD EU

Das Erweiterungs-Board netwifimicroSD-EU wird an den 120-pin MOLEX connector des gumstix verdex XL6P angeschlossen.

Technische Details:

- 10/100baseT ethernet
- microSD-Slot
- 802.11(b) und 802.11(g) WLAN-Verbindungsprotokoll
- Verbindungsanschluß : 120-pin Busheader
- Betriebsspannung: 3.5V - 6V

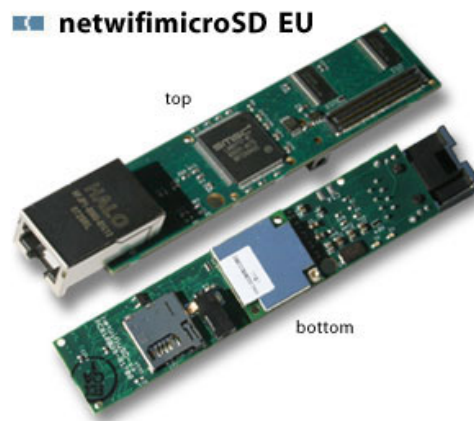


Abbildung 2.3: netwifimicroSD EU [11]

- Größe : 93mm x 20mm

Dieses Board wird primär zur drahtlosen Kommunikation zwischen dem Roboter und den Hauptrechnern eingesetzt. Wichtige Eigenschaften sind die Ethernetfähigkeit und die Unterstützung von WLAN (802.11(b) und 802.11(g) Kommunikationsnormen). Es gibt insgesamt 2 Versionen dieses Board, wobei die andere Version keinen EU-WLAN-Standard unterstützt. Zusätzlich ist auf dem Board ein Slot für eine microSD-Karte vorzufinden. Das Gumstix besitzt zwar 32 MB Flash und 128 MB Arbeitsspeicher, doch gerade bei aufwendigen Prozessen wie der Bildbearbeitung, stößt man hier schnell an die Kapazitätsgrenzen. Deshalb ist eine Aufrüstung des Speichers sinnvoll. Ein weiterer Vorteil bei der Benutzung einer Speicherkarte besteht darin, dass die Datenübertragung im Gegensatz zum konventionellen Flashen mittels serieller Schnittstelle schneller erfolgt.

Erfahrungen mit dem Gumstix

Aufgrund seiner Empfindlichkeit ist Vorsicht im Umgang mit dem Gumstix geboten. Die PG-Teilnehmer dürfen z.B. das Board nur mit angelegtem Erdungsband berühren, damit keine statische Entladung innerhalb des Boards stattfindet und somit Beschädigungen hervorgerufen werden.

Während der Inbetriebnahme des Gumstix traten diverse Probleme auf:

- Falls eine zu geringe Spannung anliegt (i.d.R. kleiner als 4-5 V), bootet das Board den Kernel nicht vollständig, weshalb immer eine ausreichende hohe Spannung angelegt werden sollte (jedoch darf sie aus Sicherheitsgründen wiederum nicht zu hoch ausfallen).
- Die WLAN-Funktionalität kann auch nach mehreren Wochen der Suche und des Probierens nicht korrekt benutzt werden. Vermutet werden neben Treiberproblemen auch evtl. Hardware-Fehler. Problematisch ist hierbei auch die Tatsache, dass Support und Erfahrungsberichte im Internet nicht vorzufinden sind.

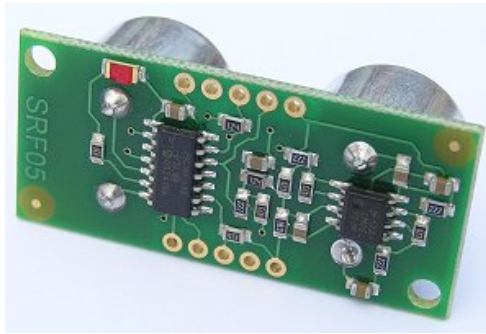


Abbildung 2.4: Der Ultraschallsensor SRF05 [8]

- Ein I2C-Anschluß war auf allen 3 Boards nicht direkt vorzufinden (wobei die Schnittstellen in der Produktbeschreibung vorhanden sind). Dieser Umstand konnte aber durch Neulöten von Pins des console-vx Boards behoben werden (siehe Kapitel 3.2 Gumstix).

2.1.2 Ultraschallsensoren

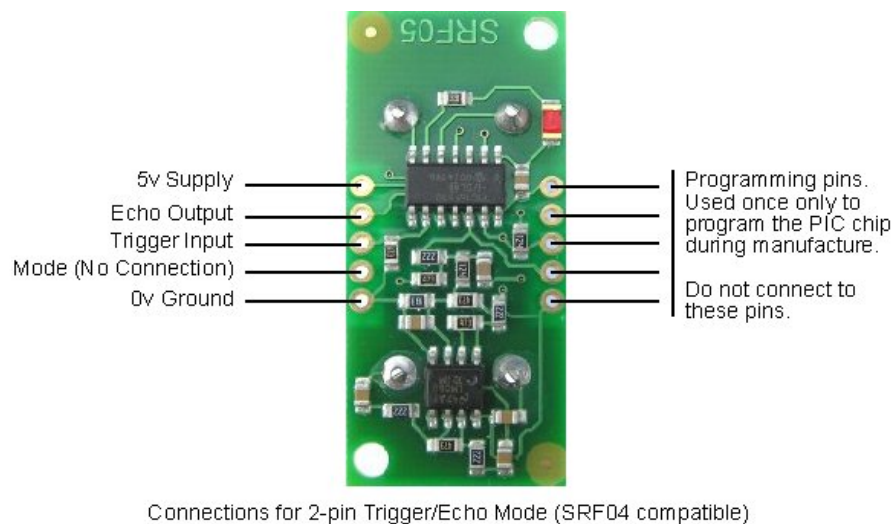
Um Hindernisse rechtzeitig erkennen zu können, hat sich die Hardwaregruppe entschieden, vier Ultraschallsensoren für ihren Staubsaugerroboter zu benutzen. Der Schall des Sensors breitet sich im Raum als Welle aus. Als Medium dient die Luft. Die Ausbreitungsgeschwindigkeit hängt von der Dichte der Luft und somit vom Ort ab. Dabei wird die Schallwelle zusätzlich an Stellen mit einer Dichteänderung teilweise reflektiert. Durch jede Dichteänderung wird ein Echo zur Ultraschallquelle reflektiert, welches als Signal wieder auffangbar und auswertbar ist. Die Zeitdifferenz zwischen aussenden und empfangen des Signals gibt bei bekannter Mediumsdichte Aufschluss über die Distanz zwischen Grenzfläche und Sensor. Somit ist eine Längenmessung entlang der Schallausbreitung möglich.

Die Firma Devantech [8] hat eine Serie von kleinen Ultraschallsensoren entwickelt. Ein vergleichbares Modul selbst zu bauen wäre ungenauer und teurer. Die Hardwaregruppe hat sich für das Modell SRF05 entschieden, siehe Abbildung 2.4. Dieses besitzt eine Reichweite von bis zu 4 Meter. Des Weiteren hat er einen zweiten Modus, in dem sowohl der Start der Messung, als auch das Ergebnis über die gleiche Leitung übertragen werden (siehe Abschnitt über Modus 2). Der Ultraschallsensor kann folglich über einen einzigen Port (Pin) gesteuert werden. Diesen Vorteil verwendet die Projektgruppe für ihre vier Ultraschallsensoren. Im Folgenden werden die beiden Modi und deren Pinbelegung näher erläutert. Ferner wird auf die Distanzberechnung und die Richtwirkung des Sensors näher eingegangen.

Modus 1

Wie in Abbildung 2.5 zu sehen ist, benutzt der erste Modus separate Trigger- und Echo-Pins, das heisst es müssen zwei Ports mit dem Mikrocontroller verbunden werden. Der SRF05 hat einen internen Pull-Up Widerstand, so dass der Modus-Pin unverbunden bleiben

kann. Ein kurzer High-Impuls, auf dem Trigger Input, von mindestens 10 Mikrosekunden löst den Messvorgang aus. Anschließend überwacht der Mikrocontroller den Echo-Pin. Die Länge des High-Impulses ist proportional zur Entfernung des Objektes. Die gemessene Zeit (gemessen wird in der Zeiteinheit μS) wird durch die Konstante 58 dividiert, wodurch die Entfernung in Zentimetern erhalten wird.



Connections for 2-pin Trigger/Echo Mode (SRF04 compatible)

Abbildung 2.5: Anschlüsse für separaten Trigger- und Echo-Pin [8]

Modus 2

Der zweite Modus besitzt einen gemeinsamen Pin für das Trigger- und Echo-Signal, siehe Abbildung 2.6. Er hat den Vorteil, dass nur ein Pin des Controllerports belegt ist, denn die Auslösung und das Messen des Ergebnisses erfolgt über den gleichen Port. Um diesen Modus zu aktivieren, muss der Modus-Pin am Ultraschallsensor mit Low (GND) verbunden werden.

Nach Senden eines Auslösesignales, ein mindestens $10\mu S$ langes High-Signal, muss der Port auf Input umgeschaltet werden. Danach wird das High-Signal des Ultraschallsensors abgewartet. Für diese Umschaltung stehen mindestens $700\mu S$ zur Verfügung. Analog zur oberen Beschreibung des ersten Modus ist die Länge des zurückgelieferten High-Impulses proportional zur Entfernung des Objektes. Indem die gemessene Zeit durch die Konstante 58 dividiert wird, wird die Entfernung in Zentimetern erhalten.

Distanzen berechnen

Die Zeitdiagramme des SRF05 werden für den jeweiligen Modus in den Abbildungen 2.7 und 2.8 gezeigt. Wie zuvor in den Abschnitten über Modus 1 und 2 erwähnt, reicht ein mindestens $10\mu S$ langes High-Signal aus, um den Messvorgang auszulösen. Der SRF05 sendet acht Zyklen hintereinander einen ultrahochfrequenten Ton, bei 40kHz, aus. Während

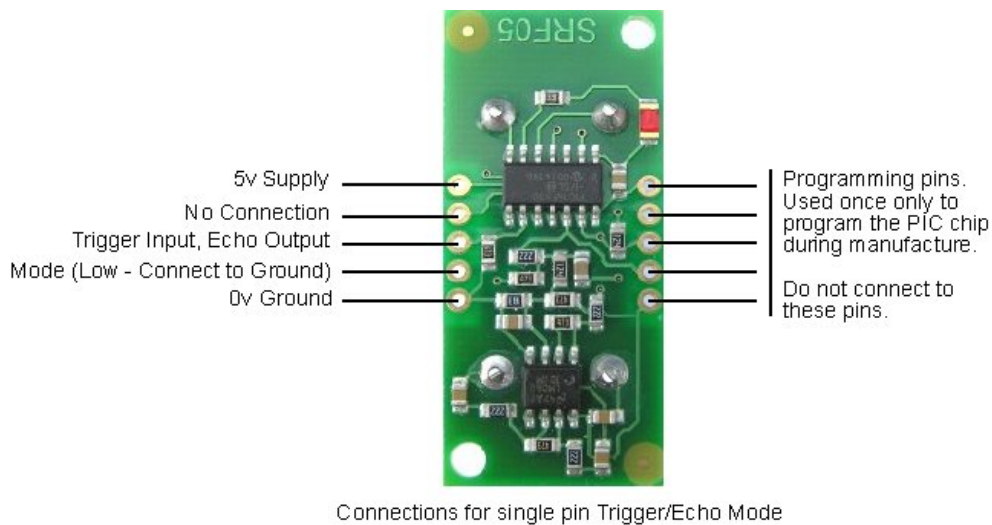


Abbildung 2.6: Anschlüsse für gemeinsamen Trigger- und Echo-Pin [8]

dieser Zeit hat er einen High-Pegel. Wird ein Echo registriert, wird der Pegel in diesem Fall auf Low gesenkt. Die Länge des High-Pegels ist proportional zur Entfernung des Objekts. Mit einer Zeitmessung lässt sich die Distanz, beispielsweise in Zentimeter, berechnen. Wird kein Echo empfangen, senkt der Ultraschallsensor den Pegel nach 30 Millisekunden auf Low. Bis zu 20 Messungen pro Sekunde sind mit diesem Ultraschallmodul möglich, das heisst alle 50 Millisekunden ist eine Messung möglich. Wichtig ist dabei, dass die Zeit des Echo-Impulses möglichst genau ermittelt wird. Je genauer dies geschieht, desto akkurater ist die Entfernungsmessung. Abschließend ist noch zu erwähnen, dass die restlichen fünf Pins, auch Programmier-Pins genannt, ausschliesslich bei der Herstellung dazu dienen, den Flash-Speicher des Chips (PIC16F630) zu programmieren. Das bedeutet, dass keiner dieser Pins verbunden werden darf.

Richtwirkung des Ultraschallsensors

Die Richtwirkung des SRF05 ist dem Datenblatt des Herstellers entnommen und ist in der Grafik 2.9 zu sehen. Die Ultraschallwellen breiten sich konisch aus. Die Richtung ist nicht änderbar.

2.1.3 Kontaktsensoren

Die Ultraschallsensoren können ein Hindernis erst ab einem Mindestabstand von 10cm messen. Zudem haben sie eine Fehlerrate. In einigen Szenarien kann es dazu kommen, dass die Sensoren ein Hindernis nicht erkennen. Für diesen Fall hat die Projektgruppe sich entschieden den Staubsaugerroboter mit fünf zusätzlichen Microschaltern, auch Schnappschalter genannt, zu versehen. Die Entscheidung fiel auf den Schalter MBF5B der Firma Hartmann [12], siehe Abbildung 2.10. Die Abbildung 2.11 zeigt eine Zeichnung des Schal-

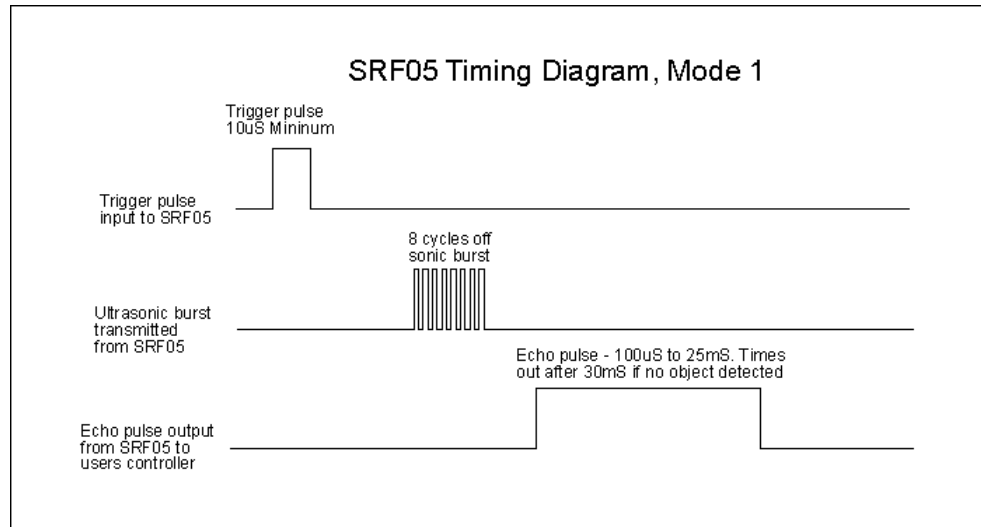


Abbildung 2.7: Zeitdiagramm vom Modus 1 [8]

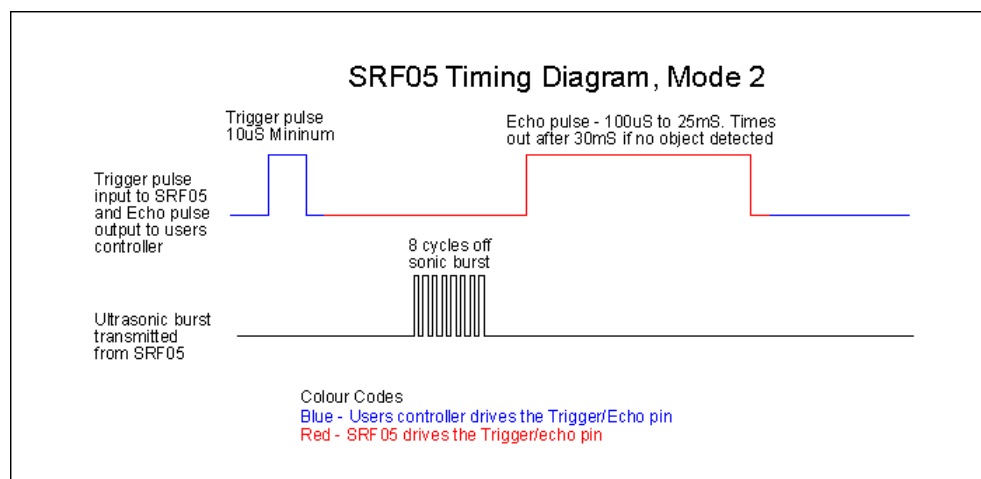
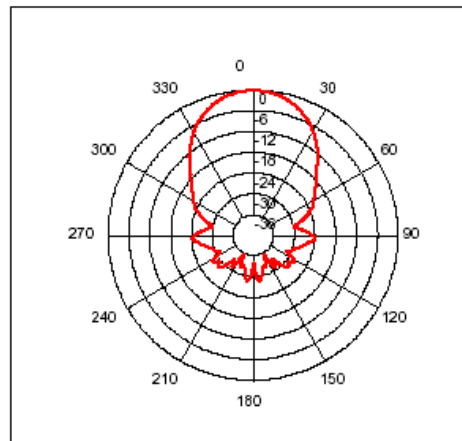


Abbildung 2.8: Zeitdiagramm vom Modus 2 [8]



ters mit seinen Anschlüssen. Der Kontakt ist in Ruhestellung zwischen COM und NC oder NO geschlossen. Durch Druck auf den Betätiger wird der Schaltvorgang ausgelöst. Dabei wird von Kontakt NC auf NO umgeschaltet. Der Kontaktabstand ist kleiner als 3mm.

Type MB

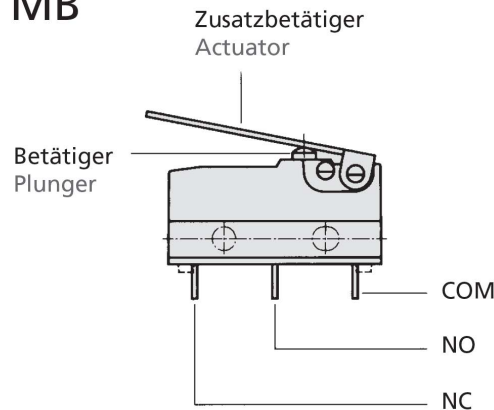


Abbildung 2.11: Anschlüsse des Kontaktsensors [12]

2.1.4 Lichtsensoren

Ein Lichtsensor ändert bei der Einwirkung von Licht seine elektrischen Eigenschaften. Der Reflex Optokoppler CNY70, siehe Abbildung 2.12, der Firma Vishay [26] wird für den Staubsaugerroboter vielfältig eingesetzt. Wie in den folgenden Abschnitten beschrieben, dient er der Projektgruppe als Radencoder und Abgrundsensoren. In der Zeichnung 2.13 ist dargestellt, dass der CNY70 in seinem würfelförmigen Gehäuse eine Infrarot-LED als Sender und einen Infrarot-Fototransistor als Empfänger besitzt. Damit kann auf kurze Entfernung das reflektierte Licht der IR-LED durch den IR-Fototransistor gemessen werden. Die Menge des reflektierten Lichts bestimmt, ob der Fototransistor mehr oder weniger leitend wird. Die Reichweite des Sensors beträgt nur wenige Millimeter. Störungen durch Fremdlichteinstrahlung sind eine mögliche Fehlerursache.

Radencoder

Mit Hilfe seiner beiden Radencoder kann der RAPTOR die zurückgelegte Wegstrecke bestimmen, eine bestimmte Strecke geradeaus fahren, sowie sich um einen bestimmten Winkel drehen. Auf der Innenseite der Räder sind Encoderscheiben aufgeklebt. Auf ihnen sind insgesamt 30 schwarze und 30 weiße Streifen im Wechsel kreisförmig angeordnet. Ein Radencoder besteht aus einem Lichtsensor, dem CNY70, der so am Roboter angebracht ist, dass er auf die Streifen der Encoderscheiben zeigt. Der Lichtsensor erzeugt analoge Signale. Sie werden für die weitere Verarbeitung durch den Mikrocontroller mit Hilfe eines Schmitt-Triggers in digitale Signale umgesetzt. Ein Schmitt-Trigger erzeugt aus einem ana-

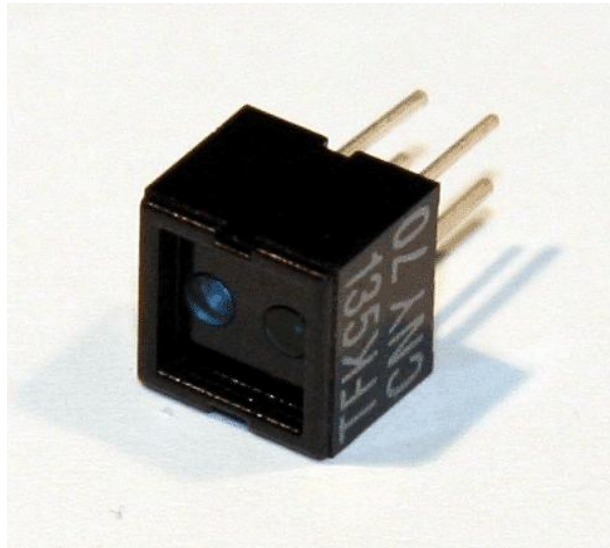


Abbildung 2.12: Der Lichtsensor CNY70 [26]

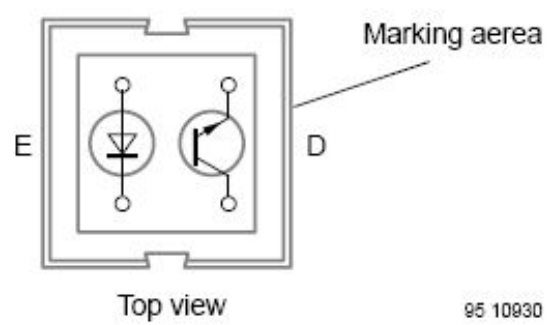


Abbildung 2.13: Steckerbelegung des CNY70 [26]

logem Eingangssignal eindeutige digitale Schaltzustände. Aufgrund dessen reicht es, wenn der Mikrocontroller nur noch die Flankenwechsel von High nach Low zählt.

Abgrundsensord

Abgründe, wie beispielsweise Treppen und Tischkanten, erkennt der RAPTOR mit Hilfe von drei CNY70, siehe Abbildung 2.12, die als Abgrunddetektoren fungieren. Sie befinden sich an der unteren Front des Staubsaugers. Wie bei dem Maussensor darf der Abstand zwischen Lichtsensor und Boden nicht mehr als 3mm betragen. Problematisch können beispielsweise Linien am Boden sein, die der Abgrundsensord fälschlicherweise als Abgrund erkennt.

2.1.5 Maussensor

Da die Radencodier und die Ansteuerung der Motoren keine hinreichende Sicherheit bei der Erkennung der Länge abgefahrener Strecken gewährleisten können, hat die Projektgruppe sich für einen Maussensor als zusätzliche Navigationshilfe entschieden. Optische Maussensoren sind heutzutage feste Bestandteile moderner Mäuse. Im Gegensatz zu Mäusen mit Rollkugel, wo die Bewegung des Benutzers mechanisch erfasst wird, erfolgt hier eine optische Erfassung der Bewegung. Prinzipiell ist ein optischer Maussensor nichts anderes als eine kleine Kamera, die ständig Bilder der erfassten Fläche aufnimmt und Änderungen mathematisch ermittelt. Somit kann eine Positionsänderung präzise berechnet werden.

Das Project c't-Bot [7] benutzt einen Maussensor, der auch sehr gut die Anforderungen für einen Staubsaugerroboter erfüllt, und wurde somit von der Hardware-Gruppe weitestgehend übernommen. Als Sensorchip wird der ADNS-2610 der Firma Agilent Technologies [1] eingesetzt. Dieser wird gemeinsam mit einer LED zur Beleuchtung des Bodens, einer Linse und einigen weiteren Bauteilen wie Quarz zur Taktgebung auf einer eigenen Platine untergebracht und von unten an den Staubsaugerroboter befestigt. Die Verbindung zu dem Mikrocontroller wird über die beiden Pins SDIO und SCLK realisiert.

Funktionsweise

Der ADNS-2610 besteht aus einem Bilderfassungssystem, einem DSP zur Bildverarbeitung und einem seriellen 2-Pin Bus Interface zur Host-Kommunikation. Der Chip verarbeitet 1512 Bilder pro Sekunde bei einer Sensorgröße von 18x18 Pixeln und einer Auflösung von 64 Graustufen. Die Auflösung beträgt 400 dpi.

Das Funktionsprinzip nennt sich 'Optisches Navigations System' und beruht darauf, dass der Bildsensor ständig mikroskopisch kleine Bilder von der Oberfläche aufnimmt, und der DSP diese Bilder ständig miteinander vergleicht. Aus den unterschiedlichen Bildern wird dann die Richtung und die zurückgelegte Strecke errechnet. Als Ausgabe stehen die aktuellen X- und Y-Werte in Registern, die über den seriellen Port abgefragt werden können.

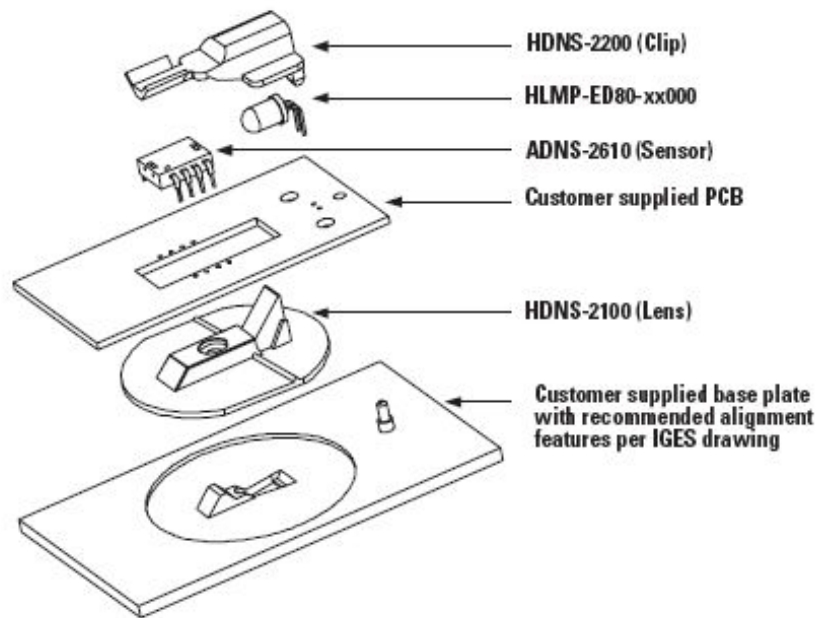


Abbildung 2.14: Bestandteile des optischen Maussensors [1]

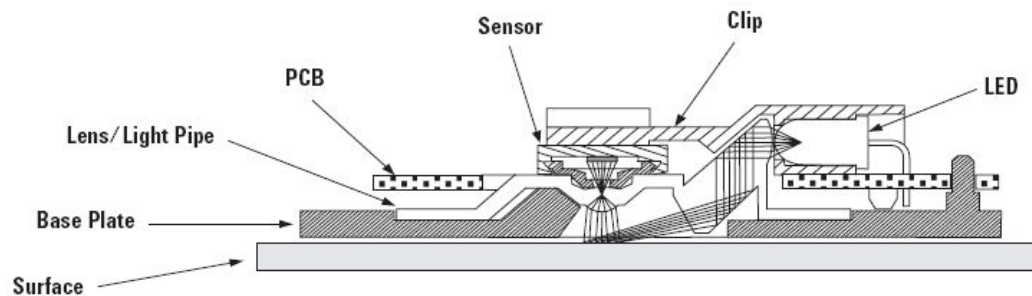


Abbildung 2.15: Aufbau und Funktionsweise der Optik des optischen Maussensors [1]

2.1.6 Kompasssensor

Die Verwendung einer Karte für die Navigation setzt eine zuverlässige Positionsbestimmung des Roboters im Raum voraus. Ein Maussensor liefert zwar die abgefahrte Strecke, kann aber insbesondere auf glatten oder spiegelnden Oberflächen ungenau sein. Aus diesem Grund hat sich die Hardware-Gruppe entschieden, die Fahrtrichtung des Roboters zusätzlich über einen Kompasssensor zu bestimmen.

Ein passendes Modul mit der Bezeichnung CMPS03 wurde bei der Firma Devantech [8] gefunden. Im Vergleich zu Kompasssensoren anderer Hersteller zeichnet sich der CMPS03 durch geringen Preis, fertigem Aufbau des Moduls und Einfachheit des Auslesens der Daten aus. Dieses Modul ist speziell für die Bedürfnisse von Robotern gestaltet worden. Es ermittelt anhand des Erdmagnetfeldes die genaue Himmelsrichtung in 0,1 Grad Schritten. Das Modul arbeitet auch in geschlossenen Räumen korrekt, was insbesondere für den Staubsaugroboter wichtig ist. Die Ausgabe der Ergebnisse kann entweder als PWM-Signal, oder über einen I2C-Bus abgefragt werden. Im Folgenden werden die beiden Betriebsarten und die Pinbelegung des Kompasses näher erläutert.

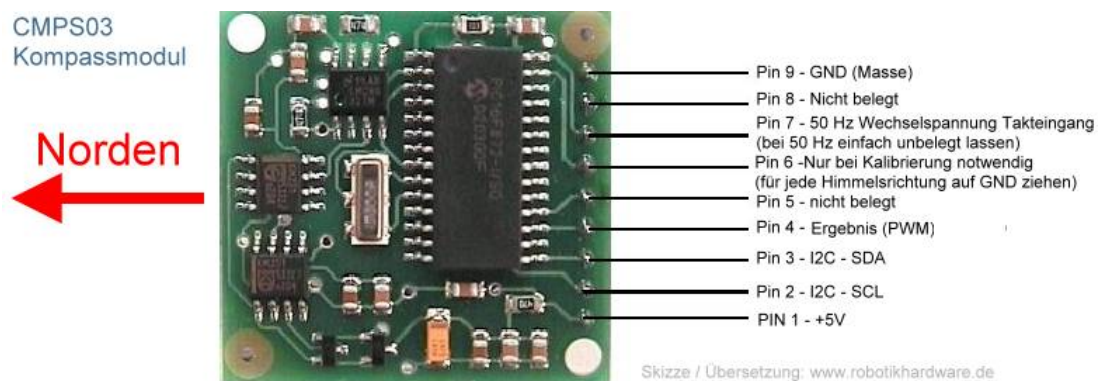


Abbildung 2.16: Pinbelegung des Kompassensors [8]

Modus 1

Das PWM-Signal an Pin 4 gibt den Kompasswert 0 bis 359,9 Grad in Form eines High Impulses aus. Die Länge des High-Impulses kann zwischen 1 Millisekunde und 36,99 Millisekunden liegen. Demnach entsprechen 0,1 Millisekunden (100us) einem Grad. Es muss somit nur die Signallänge gemessen werden, danach ist eine Umrechnung sehr einfach.

Modus 2

Der I2C-Bus wird an den Pins 2 und 3 (SDA und SCL) angeschlossen. CMPS03 besitzt keine Pullup-Widerstände wie sie für den I2C-Bus notwendig sind. Das Masterboard am I2C-Bus sollte demnach diese I2C-Leitungen mit ca. 5k bis 47k Widerständen mit +5V

verbinden. Wird der I2C-Bus bei CMPS03 nicht genutzt, so sollten Pin 2 und 3 über einen Widerstand (ca. 10k bis 47k) mit +5V verbunden sein, damit Störungen vermieden werden.

Die Kommunikation über den I2C-Bus erfolgt, wie bei fast allen Modulen, über verschiedene Register. Es stehen folgende Register zur Verfügung:

Register	Funktion
0	Software Version (Firmware Version)
1	Kompasswert - Ein Byte 0 bis 255 entspricht 0 bis 359,9 Grad
2,3	Kompasswert als Word (also 2 Byte / Low and High) Der Wert von 0 bis 3599 entspricht 0 bis 359,9 Grad.
4,5	Interne Testregister - werden nur vom Hersteller genutzt
6,7	Interne Testregister - werden nur vom Hersteller genutzt
8,9	Interne Testregister - werden nur vom Hersteller genutzt
10,11	Interne Testregister - werden nur vom Hersteller genutzt
12	Unbenutzt, liefert immer 0 zurück
13	Unbenutzt, liefert immer 0 zurück
14	Unbenutzt, liefert undefinierten Wert zurück
15	255 :startet die Kalibrierung (Justierung der Richtungen) 0 : Beendet Kalibrierung. Werte werden im EEPROM gespeichert

Tabelle 2.1: Registerbelegung des Kompassmoduls

Die Slave-Adresse ist fest auf Hex C0 (also Dezimal 192) eingestellt. Wie bei I2C üblich, wird immer erst die SLAVE Adresse und dann das abzurufende Register als Byte versendet. Anschließend erfolgt ein Lesezugriff, in dem die SLAVE Adresse +1 gesendet und danach 1 oder 2 Bytes abgerufen werden. CMPS03 unterstützt die Standard I2C Taktraten von 100 und 400 KHz. Wird mehr als 100 KHz genutzt, sollte nach Schreiben der Registeradresse eine kurze Pause von ca. 50us eingebaut sein.

Pin 1 und 9 - Stromversorgung

Das Kompassmodul benötigt lediglich 5V am Pin 1 und verbraucht im Durchschnitt ca. 15 mA. Pin 9 wird mit Ground verbunden.

Pin 6 - Kalibrierung

Damit das Modul in jeder Umgebung möglichst genau arbeitet, ist eine einmalige Justierung (Kalibrierung) empfehlenswert. Die Kalibrierung erfolgt indem das Modul exakt waagerecht in alle viel Himmelsrichtungen ausrichtet und jedes Mal dabei Pin 6 kurz mit GND verbunden wird. Das Ergebnis wird intern gespeichert und bleibt auch dann erhalten, wenn keine Spannung anliegt. Alternativ kann auch über den I2C-Bus die Kalibrierung durchgeführt werden. Dort funktioniert es auf die gleiche Weise, mit dem Unterschied, dass statt des Tasterdrucks eine 255 in das Register 15 geschrieben wird.

Pin 7 - Wechsellspannungsfelder

Die Abtastrate der Kompassposition erfolgt bei unbeschaltetem Pin 7 intern gewöhnlich mit 60 Hz. In Umgebungen mit starken Wechsellspannungsfeldern kann es wegen des dadurch erzeugten Magnetfeldes zu Ungenauigkeiten kommen. In diesem Fall kann es günstig sein die Netzfrequenz über diesen Pin mit dem Modul zu synchronisieren. Dazu muss das passende 50 Hz Taktsignal mit TTL Pegel angelegt werden. Dies kann das Ergebnis in solch schwierigen Umgebungen verbessern. In der Regel kann der Pin unbeschaltet gelassen werden.

2.1.7 Getriebemotoren

Bei der Entwicklung eines mobilen Roboters wird man nicht drum herum kommen irgendeine Art von Motor einzusetzen. Bei der Auswahl des richtigen Motors müssen mehrere Kriterien beachtet werden:

- Größe
- Strombedarf
- Steuerbarkeit
- Drehmoment

Die Größe der Motoren müssen, je nach Robotergröße ausgesucht werden, denn die Motoren nehmen auf der einen Seite wichtige Flächenressourcen für andere Bauteile weg. Auf der anderen Seite dürfen die Motoren nicht zu schwach sein, dadurch wäre der Roboter nicht mehr in der Lage sich fortzubewegen.

Klare Schrittweiten und/oder präzise Geschwindigkeiten ermöglichen es den zurückgelegten Weg relativ exakt zu bestimmen, sowie die Fahrt grader Strecken zu realisieren. Der Motor sollte auch präzise halten können, wenn die Abstandsensoren einen Abgrund erfassen, um das Runterfallen des Roboters zu vermeiden.

Das Drehmoment der Motoren bestimmt die Fortbewegung. Ein geringer Drehmoment in Kombination mit hoher Last führen zu einer Überlastung des Motors bzw. Motortreibers. Es existieren Motortreiber (L293D), die bei Überhitzung abschalten. Dadurch besteht aber das Problem, dass der Roboter ständig stehen bleibt.

Für fahrende Roboter kommen nur zwei Motoren in Frage, und zwar Getriebemotoren oder Schrittmotoren. Im Roboterprojekt werden Getriebemotoren eingesetzt, da sie zu den einfachsten, kleinsten und universellsten Motoren zählen. Diese Motoren eignen sich besonders gut zur Konstruktion von flinken wie auch langsamen Robotern. Es können aber nur Roboter bis zu 5kg Gewicht angetrieben werden. Die Motorsteuerung für den Getriebemotor ist relativ einfach. Es gibt zwei Anschlüsse, welche mit jeweils unterschiedlicher Polarität belegt werden, um den Motor laufen zu lassen. Je nach Richtung des Stromflusses, dreht sich der Motor entweder vorwärts oder rückwärts. Die Getriebemotoren können

abhängig von ihrer Belastung und dem Untergrund (Fliesen, Teppich, Laminat, etc.) in ihrer tatsächlichen Umlaufgeschwindigkeit voneinander abweichen. Um sicherzustellen, dass zwei verschiedene Getriebemotoren gleich oft drehen, müssen externe Sensoren eingesetzt werden, die die Radumdrehung ständig überwachen und die Information an einen Mikrocontroller senden. Trotz dieser Nachteile werden Getriebemotoren eingesetzt, auch wenn die Fahrt gerader Strecken zu einer Herausforderung wird. Um zu verstehen warum der Schrittmotor nicht in Frage kommt, muss zunächst das Prinzip des Schrittmotors verstanden werden.

Die Schrittmotoren sind grundlegend anders konzipiert als die Getriebemotoren. Der Rotor besteht meist aus einem Permanentmagneten, der durch Anlegen einer Spannung an den Spulen bewegt wird. Jeder Motor enthält zwei Spulen, welche wiederum jeweils zwei Anschlüsse besitzen, und bei angelegter Spannung den Motor um einen Schritt fortbewegen. Die Fortbewegung des Rotors pro Schritt ist relativ präzise, (die Toleranz liegt höchsten bei 5 Prozent). Der Schrittwinkel gibt dabei an, wieviele Schritte notwendig sind, damit das Rad einmal um die Achse dreht, z.B. werden bei einem Schrittwinkel von 1.8 Grad 200 Schritte benötigt. Der Schrittwinkel liegt, je nach Bauart des Motors, in der Regel zwischen 1.8 Grad und 18 Grad. Es kann aber über das Getriebe nach belieben verkleinert werden. So ist eine hohe Präzision und Gleichläufigkeit verschiedener Motoren erreichbar. Der Haltemoment eines Motors ist etwa gleich der Größe des Drehmomentes. Der Schrittmotor ermöglicht es mittels der Motordrehschrittzahl eine präzise Angesteuerung, d.h. der Motor muss z.B. nicht kurzgeschlossen werden, um zu bremsen.

Im Vergleich zu den Getriebemotoren ist die Ansteuerung der Schrittmotoren aufwendiger. Der Schrittmotor benötigt vier Anschlüsse damit der Rotor einen Schritt ausführt. Außerdem müssen spezielle Folgen von Polaritätsmustern abgearbeitet werden. Die Anschaffungskosten sind viel grösser als beim Getriebemotor, denn zwei Getriebemotoren können an einem Motortreiber (z.B. L293D) angeschlossen werden, während jeder Schrittmotor einen eigenen Treiber beansprucht. Im Projekt hat sich herausgestellt, dass die präzise Ansteuerung, die der Schrittmotor gewährleistet, nicht in diesem Umfang benötigt wird, da mittels anderer Sensoren ständig der aktuelle Standort des Roboters neu berechnet und die Fahrt anschließend fortgesetzt wird. Außerdem besteht die Gefahr, dass der Schrittmotor Schritte verlieren kann. Es ergibt sich das selbe Problem wie beim Getriebemotor und es müsste ein zusätzlicher Sensor eingebaut werden, um den Effekt wieder auszugleichen. Der Schrittmotor verliert auch mit steigender Drehzahl an Kraft, also ist der Motortyp nicht für schnelle Roboter geeignet. Ausserdem müsste, wenn ein Schrittmotor eingesetzt wird, ein geeignete Getriebewahl getroffen werden, der die Motorkraft auf die Antriebsräder realisiert. Ohne Getriebe hätten wir folgende Probleme:

- Ohne Getriebe hätte ein Schrittmotor mit einem Schrittwinkel von 18 Grad eine zu ungenaue Bewegung.
- Die am Rad einwirkende Kraft ist zu groß, der Motor würde zu stark belastet.



Abbildung 2.17: Ein ATmega32 in in PDIP-Bauform

2.1.8 Mikrocontroller

Obwohl der Gumstix selbst auch mit diversen Ein- und Ausgängen für digitale Signale ausgestattet ist, haben wir uns entschieden, die Sensordatenerfassung und Motoransteuerung auf einen Mikrocontroller auszulagern. Abgesehen davon, dass wir hierbei erheblich mehr Einfluss bei zeitkritischen Abfragen haben, ermöglicht uns dies eine bessere Trennung zwischen der Hardwareansteuerung und Aufgaben wie Routenplanung und Kartenerfassung. Zudem ist im Fehlerfalle ein Mikrocontroller erheblich günstiger und einfacher auszutauschen als das Linux-Board.

Schon sehr schnell limitierten wir unsere Auswahl auf die ATmega-Baureihe der Firma Atmel. Die Mikrocontroller dieser Baureihe haben den Vorteil, dass Prozessor und Speicher in einem Chip integriert sind. Zudem werden sie heutzutage in vielen Projekten eingesetzt; von daher existiert für viele Sensoren bereits umfangreiche Dokumentation und gut verständlicher Beispielcode. Nicht zuletzt wird der komplette c't-Bot über einen einzigen ATmega-Mikrocontroller angesteuert, was ein guter Hinweis für die Leistungsfähigkeit dieser Mikrocontroller ist.

Innerhalb dieser Baureihe fiel unsere Entscheidung dann auf den ATmega32. Dieser unterstützt eine Taktrate von 16 MHz. Dank seiner guten RISC-Architektur können die meisten Befehle innerhalb eines Taktes abgearbeitet werden und für jeden Einzelfall kann eine Ausführungszeit von etwa 62,5 nS angenommen werden - das reicht dann auch für die schnellsten unserer Sensoren aus. Weiterhin hat er einen integrierten 32 kB Flash-Speicher. Dieser kann "In-System" programmiert werden, was gerade beim späteren Debugging ein großer Vorteil sein kann - zusätzliches Aus- und Einbauen des Mikrocontrollers entfällt.

Ein weiteres Argument für den ATmega32 - im Vergleich zu anderen Mikrocontrollern -

sind die 32 I/O-Pins (verteilt auf 4 Ports), die auf der einen Seite frei in der Programmierung verwendet werden können, auf der anderen Seite aber auch auf verschiedene, hilfreiche Zusatzfunktionen umgeschaltet werden können. So ist es beispielsweise möglich, an bis zu vier verschiedenen Ausgängen ein PWM-Signal bereit zu stellen, ohne diese Ausgänge ständig softwaregesteuert umschalten zu müssen.

Weitere technische Details befinden sich bei der Hardwarebeschreibung und in den Programmierbeispielen.

2.1.9 Kamera

Kameras werden beim Roboter zu 2 Hauptzwecken eingesetzt:

- Durch Analyse der Bodenstruktur soll erkannt werden, ob die Fläche, über die der Roboter gerade fährt, verschmutzt ist.
- Zusätzlich Analyse aufgenommener Bilder der Umgebung sollen in Verbindung mit den Sensordaten dem Roboter dabei helfen, sich seiner Umgebung noch besser bewusst zu sein.



Abbildung 2.18: Logitech SweetPea QuickCam Express

Moderne Webcams, wie sie heutzutage für Videokonferenzen eingesetzt werden, zeichnen sich durch kompakte Größe und akzeptabler Auflösung aus und erzeugen auch in Echtzeitumgebungen akzeptable Ergebnisse (leistungsfähige Hardware vorausgesetzt). Der Roboter selbst muss nicht dauerhaft flüssige Videobilder erzeugen, stattdessen genügt es, in regelmäßigen Abständen Bilder der Umgebung/Bodenfläche aufzunehmen und zu analysieren. Für erste Testbilder in einer Linuxumgebung wurde eine verhältnismäßig billige Webcam der Marke Logitech SweetPea QuickCam Express von der Firma Logitech [16] gekauft. Lauf Herstellerangaben gelten als minimale Systemvoraussetzungen:

- Pentium III - 700 MHz
- RAM 128 MB
- HD 200 MB
- 28.8 KBit/s Internetverbindung

Jedoch muss bedacht werden, dass diese Systemvoraussetzungen wahrscheinlich (unter einem Windows-System mit installierter Logitech-Software) für den Echtzeiteinsatz gelten, d.h. selbst die angegebenen 600 MHz des Gumstix sind ausreichend zur Inbetriebnahme der Kamera. Als Betriebssystem wird Windows 2000/XP empfohlen, allerdings wurde die Kamera bereits erfolgreich unter Linux am LS14-Rechner in Betrieb genommen (Mithilfe des KDE-Instant-Messengers Kopete). Auch die Bilder mit 320 x 240 Pixel Auflösung sind von akzeptabler Qualität (wenn auch mit minimalen Abstrichen in der korrekten Farbdarstellung. Dies könnte auch ein Problem von Linux sein). Da die Kamera einen standardisierten USB-Anschluß besitzt und das Gumstix-Board nur über MiniUSB-Slots verfügt, muss zur Überbrückung ein Adapter samt Verlängerungskabel eingesetzt werden. Diese Hilfskonstruktion beeinträchtigt jedoch nicht die Funktionalität. Interessant ist zu erwähnen, dass unter dem Gumstix Bilder von besserer Qualität erzeugt werden als unter der Linuxumgebung des Hauptrechners.

Die Installation der Webcam unter Linux verlief nicht unproblematisch, schließlich wurde die mitgelieferte Logitech-Software für den Einsatz unter Windows konzipiert und zeigte keine Funktionstüchtigkeit unter Linux. Deshalb musste im Internet nach selbstgeschriebenen Treibern gesucht werden. Glücklicherweise ist die Treiberentwicklung für Webcams unter Linux mittlerweile weit fortgeschritten, weshalb passende, funktionstüchtige Treiber ohne großen Aufwand gefunden wurden.

Obwohl die Kamera bereits voll einsatzfähig ist, wird innerhalb der Projektgruppe diskutiert, ob die Hinzunahme einer weiteren, qualitativ hochwertigeren Webcam sinnvoll ist. Schließlich sind die Muster z.B. bei der Schmutzerkennung mit einer geringen Auflösung wie 320x240 Pixel schwer zu analysieren.

2.2 Software

Auch wenn auf viele bereits existierende Softwaresysteme zurückgegriffen werden konnte, waren nicht für alle Anforderungen fertige Lösungen vorhanden. Im Folgenden wird beschrieben, in welchen Bereichen vorgefertigte Anwendungen zum Einsatz kamen und an welchen Stellen eigene Software erstellt werden musste. Dies stellt jedoch nur einen kurzen Überblick dar, welcher einen Eindruck über unsere Vorüberlegungen zu diesem Thema geben soll. Detaillierte Beschreibungen finden sich später in diesem Dokument (Kapitel 4) an den Stellen, an denen auf die konkrete Umsetzung eingegangen wird.

Vorweg sei noch erwähnt, dass bei der verwendeten Programmiersprache die Entscheidung auf C/C++ fiel. Diese bietet zum einen, aufgrund seines objekt-orientierten Ansatzes,

eine gute Basis für parallele Arbeiten an unseren Softwareprojekten. Zum anderen läßt sich eine effiziente hardwarenahe Implementierung erreichen. Außerdem gibt es noch einen weiteren Grund über den Rahmen der Projektgruppe hinaus, welche für C++ spricht. Diese Sprache hat sich in allen Bereichen der Softwareentwicklung etabliert. Es existieren umfangreiche Dokumentationen zu diesem Thema, was uns dabei behilflich sein wird, guten Code zu schreiben, der sich leicht warten und korrigieren läßt.

2.2.1 Buildroot

Bei Buildroot handelt sich um ein Software-System, welches zur Unterstützung der Entwicklung von eingebetteten Systemen aufgebaut wurde. Es kommt bei dem von uns verwendeten Gumstix zum Einsatz. Neben der Bereitstellung des Betriebssystems werden von Buildroot auch Bootloader, Dateisysteme und diverse Cross-Compiler bereitgestellt. Diese können ganz einfach über textbasierte Konfigurationsoberflächen konfiguriert werden. Über dem hinaus werden auch alle wichtigen Treiber für verschiedenste Hardwarekomponenten zur Verfügung gestellt. Dies umfasst Schnittstellen, die I^2C , USB, WLAN, Ethernet, und so weiter. Auch die Paketverwaltung des Betriebssystems, welche verschiedene Softwarelösungen umfasst, wird vorgehalten. Hier finden sich Programme, wie zum Beispiel SSH-Server, Webserver, Dateisystemtools und diverse Bibliotheken.

Die komfortable Möglichkeit, das Potential des Gumstix voll aus zu nutzen, hat zu der Entscheidung geführt, das Buildroot System zu nutzen. Trotz anfänglicher Probleme hat sich diese Entscheidung als richtig heraus gestellt, da wir auf ein vollwertiges Linux Betriebssystem zurückgreifen konnten und unsere Erfahrungen in diesem Bereich auch auf die Arbeit in der Projektgruppe anwendbar waren. Dies hat uns sehr dabei geholfen grundlegende Probleme schnell und effizient zu lösen.

2.2.2 Player-Stage

Um unsere Fortschritte unter einer standardisierten Umgebung testen zu können haben wir uns dazu entschieden einen Simulator für den Roboter zu nutzen. Hier kam es vor allem darauf an, ein System zu finden, welches neben allen von uns verwendeten Sensoren auch eine solide Community aufweisen kann. Letzteres ist von großer Bedeutung, wenn eigene Plugins oder speziellen Einstellungen von Nöten werden. In solchen Fällen steht eine breite Zahl von potentiellen Ansprechpartnern zu Verfügung. Player-Stage wurde unseren Anforderungen voll gerecht. Es handelt sich hierbei um eine Kombination aus zwei Systemen. Zum einen den eigentlichen Simulator, welcher den Namen Stage trägt. Dieser dient dazu eine künstliche Umwelt zu erschaffen und darauf basierend die simulierten Sensoren mit Eindrücken zu versorgen. Hierbei wird ein Simulationsserver gestartet, der über geeignete Clients angesprochen wird. Der zweite Bestandteil von Player-Stage, der erwartungsgemäß Player heißt, stellt eine API zur Verfügung, welche die simulierte Hardware zugänglich macht. Dies wird über eine Client-Software realisiert, welche den bereits erwähnten Server von Stage anspricht. Die Kombination dieser beiden Komponenten befähigt uns Software für unseren Roboter zu entwickeln und testen, ohne auf die von uns verwendete Hardware

zurückgreifen zu müssen. So können mehrere Teams gleichzeitig Bestandteile unserer Robotersteuerung entwickeln.

Hierzu muss jedoch eine Vorbedingung erfüllt werden: der Zugriff auf den Simulator und unseren Roboter muss auf die exakt gleiche Art und Weise möglich sein. Hierzu haben wir eine eigene API definiert, welche die Hardware unseres Roboters abstrahiert. Im Folgenden wird diese näher betrachtet. Diese API muss nun auch den Simulator abstrahieren, was bereits umgesetzt wurde. Erste Tests der Funktionsfähigkeit stehen noch aus, aber wir sind zuversichtlich, dass wir auf dieser Basis mit hoher Geschwindigkeit unsere Softwareentwicklung vorantreiben können um am Ende ein leistungsfähiges System bereitstellen zu können.

2.2.3 Application Programming Interface

Wie bereits erwähnt, haben wir eine API definiert, um uns bei vielen Aspekten der Softwareentwicklung Schwierigkeiten zu ersparen. Diese hat zur Aufgabe die Hardware unseres Roboters zu abstrahieren. Dies trennt auf einfache Art und Weise die hardwarenahen und hardwarefernen Softwarebestandteile. So kann die zugrundeliegende Hardware effizient angesprochen werden und Eigenheiten dieser werden schon auf einer sehr frühen Ebene ausgeglichen. Sie wird bei der Fehleranalyse von großem Vorteil sein, da solche Aspekte nicht mehr betrachtet werden müssen. Zudem wird die API dabei behilflich sein die Entwicklung der höheren Funktionen parallel vor zu nehmen. Dies wird durch den über die API ansprechbaren Simulator (siehe 4.4) erreicht.

In Abbildung 4.18 ist die von uns entwickelte API zu sehen. Wir haben uns bei dem Design bemüht, uns auf wenige Funktionen zu beschränken. Es sollten Abfragen von Werten und das Setzen von Statusvariablen im Vordergrund stehen. Daher stellt die Implementierung nur eine Aufbereitung der Daten dar, welche von der Hardware geliefert werden. Das heißt, dass zum Beispiel keine zwei Sensorenwerte kombiniert werden. Dieses Vorgehen wird erst in den höheren Funktionen realisiert. Somit kann gesagt werden, dass in der Richtung von der Hardware zur Software nur die reinen Rohdaten aufbereitet, aber nicht modifiziert werden. In der anderen Richtung haben wir dagegen Wert auf mehr abstraktere Funktionen gelegt. So wird zum Beispiel in den höheren Funktionen kein Wissen darüber nötig sein, wie die genaue Ansteuerung der Motoren realisiert wird. Dies wird von der Implementierung der API umgesetzt.

Diese Kombination bringt zwei große Vorteile. Zum einen gehen auf der Sensorenseite keine Informationen verloren, da die Auswertung von spezialisierten Algorithmen übernommen wird. Die Aktoren dagegen bekommen für ihre Arbeit alle Informationen die sie brauchen in einer möglichst komprimierten Form. Des Weiteren wird die Geschwindigkeit des Gesamtsystems erhöht, da die Sensordaten sehr schnell zur Verfügung stehen und die Aktuatoren durch die hardwarenahe Implementierung der API effizient angesteuert werden können. Dies alles wird dazu beitragen, ein Höchstmaß an Effizienz bei der Nutzung der verwendeten Hardware zu erreichen.

2.2.4 Umgebungsmodell

Auch wenn bereits verschiedene Softwaresysteme zur Steuerung eines mobilen Roboters existieren, läßt es sich nicht vermeiden eigene Software zu programmieren. Dies hat mehrere Gründe. Zum einen ist gerade das Feld der Robotik noch nicht so weit fortgeschritten, dass dort Standardlösungen für eine Vielzahl von Aufgaben existieren. Auch ist kaum ein Forschungsfeld wie dieses so komplex in der Anzahl der möglichen Anwendungen. Da auch bei der Entwicklung des RAPTOR versucht wird besondere Aspekte zu berücksichtigen, blieb es nicht aus, maßgeschneiderte Software zu entwickeln, die diesen Anforderungen entspricht. Hierbei ist vor allem die interne Repräsentation der Welt zu nennen. Dieses Abbild der realen Welt ist das Herzstück des RAPTOR und benötigte deshalb auch eine besondere Aufmerksamkeit. Es wurden hier verschiedene Anforderungen formuliert. Zum einen muss eine Anpassung an die begrenzte Rechenleistung der verwendeten Hardware möglich sein, zum anderen darf dies nicht die Effizienz des Umgebungsmodell verringern. Hier war es notwendig eine eigene Entwicklung vorzunehmen. Es wurde zu Beginn der Projektgruppe festgelegt, eine Kombination aus zwei Kartentypen zu nutzen, welche jede für sich spezifische Vorteile aufweist. Zum einen eine Gitterkarte, welche den schnellen Zugriff auf einzelne Elemente der Karte möglich machte. Und zum anderen ein Voronoi-diagramm zur Routenberechnung, welche sich dadurch auszeichnet, dass sie sehr gut eine dynamische Route festlegen kann, die einfach an dynamische Veränderungen der Umgebung angepasst werden kann. Wie in Kapitel 4 ausführlich beschrieben wird, stellten sich manche der gemachten Annahmen als falsch heraus. Gerade das Voronoidiagramm wurde in seiner Komplexität unterschätzt. Eine eigene Neuentwicklung wurde bereits zu einem frühen Zeitpunkt verworfen. Statt dessen wurde versucht eine bestehende Bibliothek an die Bedürfnisse des RAPTOR anzupassen. Dies misslang leider, was schlussendlich dazu führte, dass die Architektur des Umgebungsmodell überdacht wurde. Das Voronoidiagramm weist eine zu hohe Komplexität auf, als das realistisch davon ausgegangen werden kann, in der zur Verfügung stehenden Zeit, eine fehlerfreie Implementierung hin zu bekommen. Auch sind die Annahmen über die Laufzeit der Berechnung eines Voronoidiagramms unter falschen Vorraussetzungen entstanden. Es werden viele komplexe Rechenoperationen nötig, die von der genutzten Hardware nicht effizient umgesetzt werden kann.

Die Kartenarchitektur wurde also einzig und alleine auf die Gitterkarte aufgebaut. Diese bietet durch ihren schnellen Zugriff eine bessere Basis zur Routenberechnung. Diese wird durch verschiedene Techniken der Bildverarbeitung realisiert, die aber rein auf ganzzahligen Werten und Operationen aufbauen, die von der verwendeten Hardware problemlos bewältigt werden können. Auch handelt es sich um einfacher zu implementierende Algorithmen, welche durch verschiedenste Techniken zusätzlich optimiert werden können. Auch die Fehleranfälligkeit sinkt deutlich aufgrund der geringeren Komplexität.

2.2.5 Mikrocontroller

In der Welt der Robotik findet sich kaum ein System, welches nur von einer zentralen Einheit gesteuert wird. Es kommen immer stärker Mikrocontroller zum Einsatz, die kleinere

Aufgaben schneller und kostengünstiger erledigen können als reine CPUs. Dies hat mehrere Vorteile. Die Hauptrecheneinheit kann zum Beispiel von niedrigeren Arbeiten entlastet werden. Diese niedrigeren Aufgaben können mit Mikrocontrollern sogar sehr oft besser erledigt werden, als mit einer CPU die gleichzeitig noch ein ganzes Betriebssystem zu verwalten hat. Dies führt zu schlechteren Ansprechzeiten, welche beim Auslesen eines Sensors die Zuverlässigkeit der gemessenen Daten geradezu eliminieren. Hier bilden die Mikrocontroller die solide Grundlage, auf die bei Bedarf über entsprechende Bussysteme zurückgegriffen werden kann.

Da es sich hierbei um eine sehr technische Angelegenheit handelt, welche weniger durch die verwendete Software bestimmt wird, wird an dieser Stelle nicht weiter auf diesen Aspekt eingegangen. Hiermit schließt die Beschreibung der Grundlagen der Software und es folgt die detaillierteren Beschreibungen der Arbeit an dem RAPTOR.

3 Hardware

Mit Hilfe der Grundlagen aus dem zweiten Kapitel spezialisiert sich dieses Kapitel auf die Realisierung der Hardware des autonomen Staubsaugerroboters. Im Folgenden wird die Testumgebung der Projektgruppe beschrieben. Diese bewährte sich besonders am Anfang des Wintersemesters 2007/2008. Des Weiteren werden Einzelheiten zum ATmega32 erläutert. Ferner wird ein Einblick über die Implementierung der Hardware erfolgen. Bevor die Realisierung der Hauptplatine des Roboters beschrieben wird, wird erklärt wie die Hardwaregruppe ihre Platinen hergestellt hat. Abschliessend wird die Karosserie beschrieben. Dabei wird auf die Platzierungen der einzelnen Hardwareelemente und die entsprechenden Befestigungsmethoden eingegangen.

3.1 Testumgebung

Zu Beginn der Projektgruppe, als die Teilnehmer die ersten Erfahrungen mit der Hardware, verschiedenen Sensoren und der Programmierung gemacht haben, waren verschiedene Versuche auszuführen.

Da es nicht praktikabel war, für jeden Versuch und jeden Aufbau endgültige Lösungen zu bauen, die dann natürlich auch bei jedem Fehler hätten neu erstellt werden müssen, wurde eine kleine Testumgebung zusammen gestellt, auf der schnell und relativ einfach experimentiert werden konnten.

3.1.1 Steckbrett

Um nicht für jeden Testaufbau direkt eine Lochrasterplatine oder etwas ähnliches zusammen zu löten, wurde zu Beginn mit einem Steckbrett (Abbildung 3.1) gearbeitet. Dort ist es relativ einfach möglich, kleinere Schaltungen einfach zusammen zu stecken und Verbindungen durch Drahtbrücken zu erzeugen. Die Löcher haben einen handelsüblichen Rasterabstand, so dass sich viele Bauteile direkt oder mit nur wenig Aufwand einstecken lassen.

Interne Verbindungen der Steckleisten

Die eigentlichen Steckleisten sind spaltenweise miteinander verbunden. (Abbildung 3.2) Mit dem Raster der Beschriftung heißt das also:

A1-E1; F1-J1; A2-E2; F2-J2;...

Die beiden Steckleisten sind untereinander nicht verbunden. Auch sind Pins in verschiedenen Spalten nicht verbunden.

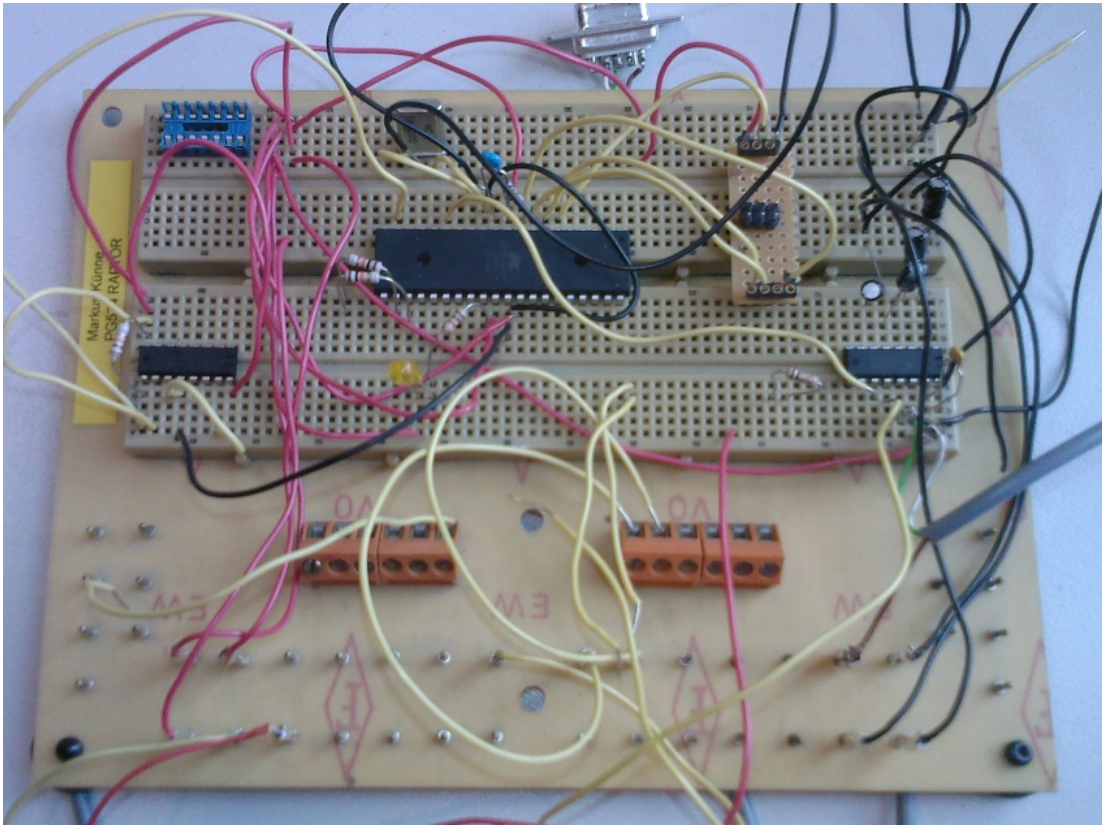


Abbildung 3.1: Das Steckbrett unserer Testumgebung - mit einem Testaufbau

Konventionen

Da auf einem solchen Steckbrett die Verbindungen in dem Moment geschaffen werden, in dem sie gebraucht werden, geht sehr schnell der Überblick verloren (siehe auch Abbildung 3.1). Es wurde versucht, diesen Effekt durch ein wenig Struktur etwas einzudämmen. Insbesondere die Verwendung verschiedener Kabelfarben sollte für etwas Übersicht sorgen. Da im Hardwarelabor eher wenig verschiedene starre Kabel bereit lagen, war nur wenig Differenzierung möglich:

- Schwarze Drahtbrücken wurden verwendet, wenn an beiden Enden auf jeden Fall GND-Pegel anliegen sollte.
- Rote Drahtbrücken wurden verwendet, wenn an beiden Enden auf jeden Fall Spannung anliegen sollte (meist 5V, beim Motortreiber wurden aber auch höhere Spannungen genutzt.)
- Gelbe Drahtbrücken wurden für alles andere verwendet: Signale, Eingänge, Ausgänge, etc.

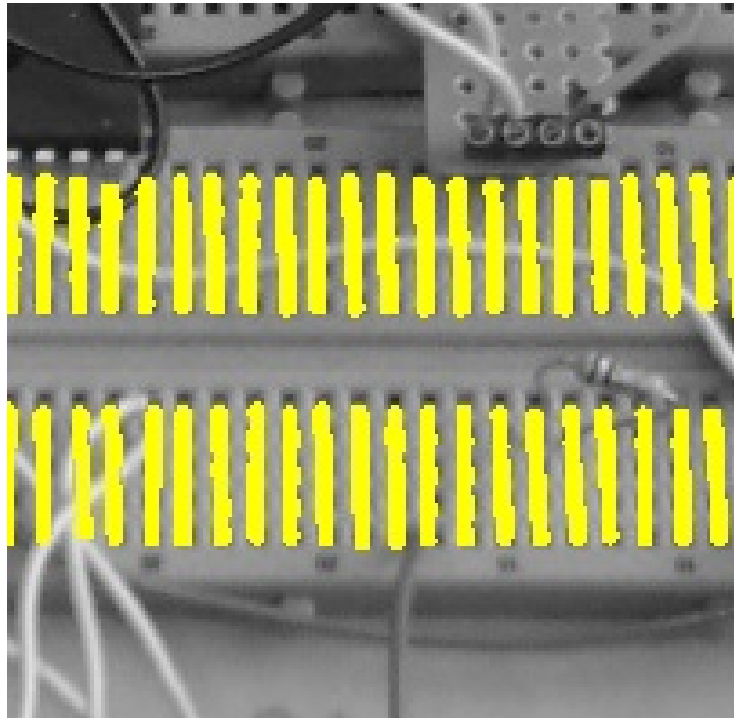


Abbildung 3.2: Die versteckten Verbindungen am Steckbrett

Zusätzlich wurde versucht, “logische Einheiten” zu schaffen, also vergleichbare Bauelemente nah beieinander unterzubringen. Auch die Verwendung von ganzen Ecken nur als Strom-Bus bzw. GND-Bus scheint übersichtlicher, als ein endloses “Daisy-Chaining”.

Interne Verbindungen an den Lötstiften

Auf dem unteren Teil des Testbretts befinden sich noch einige Lötstifte und Klemmen. Zum größten Teil ist hier jeweils nur die Klemme auf zwei Stifte gelegt worden, die äußerste Klemme ist jedoch direkt mit vier Stiften verbunden (womit sie sich für die ständig benötigte Stromversorgung und die Masse gut eignet), die sieben äußersten Pinne sind nur unter sich verbunden.(Abbildung 3.3)

3.1.2 BreakOutBox

“BreakOutBox” ist ein umgangssprachlicher Name für eine kleine Adapterbox, die einige Ein- und Ausgänge bereitstellt und relativ einfach am PC anzusteuern ist (Abbildung 3.4). Der Anschluss erfolgt über einen freien USB-Port am PC.

Die Produktbezeichnung für die Box ist “PMD-1208LS”, inzwischen wurde sie jedoch von der Herstellerfirma umbenannt und ist somit eher unter der Bezeichnung “USB-1208LS” zu finden. Die Herstellerfirma ist “Measurement Computing” [6], in Deutschland wird die Box von der Firma “Plug-In Electronics GMBH” [10] vertrieben.

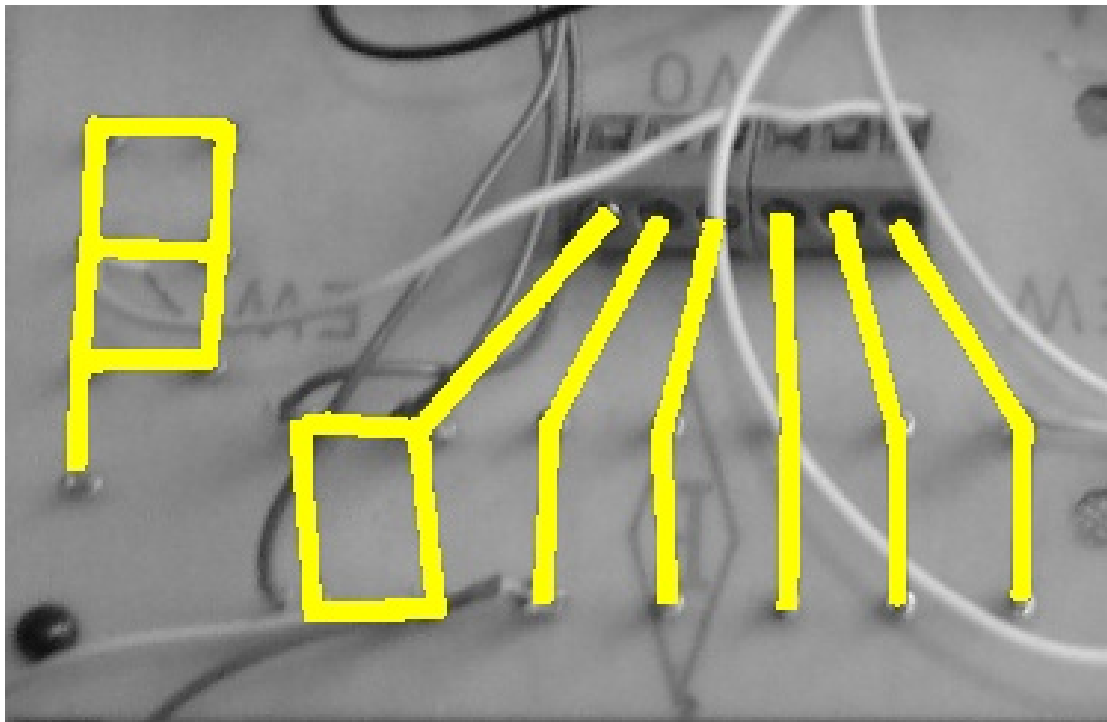


Abbildung 3.3: Die versteckten Verbindungen an den Lötstiften, rechte Seite analog



Abbildung 3.4: Die “Breakout-Box” [18]

Das Gerät verfügt über acht analoge Eingänge, zwei analoge Ausgänge sowie 16 digitale Pins. Die digitalen Pins lassen sich einzeln als Eingang oder Ausgang konfigurieren. Die Stromversorgung erfolgt direkt über den USB-Port des PCs, eine externe Spannungsquelle wird nicht benötigt.

Windows und Linux

Laut einigen Web-Berichten soll dieses Gerät auch unter Linux funktionieren, allerdings gab es dort einige Komplikationen mit dem HID-Treiber. Auch ist unter Linux für Zugriffe auf die Box eine Anmeldung als *root* erforderlich. Letzendlich wurde für den Zugriff auf die Box ein Windows-Rechner aufgesetzt, über den dann auch mit der Hersteller-Software auf die Box zugegriffen werden konnte.

Pinbelegung der Box

Eine grobe Übersicht der Klemmenbelegung der Box befindet sich in Tabelle 3.1. Die Details können im Handbuch oder auf der Hersteller-Webseite eingesehen werden.

Grundsätzlich sind Klemmen 01-20 für den analogen Teil des Gerätes zuständig, während sich auf Klemmen 21-40 die digitalen Ports befinden.

30	P5V Ausgang	Kann verwendet werden, wenn externe Sensoren eine Stromversorgung brauchen, gibt im Wesentlichen die 5V vom USB-Bus des Computers weiter, sollte also nicht überlastet werden.
1, 6, 9, 12, 15, 17, 19, 29, 31, 40	GND	Es ist davon auszugehen, dass intern nur einmal Ground verwendet wird und diese Kontakte alle durchverbunden sind.
1, 2, 4, 5, 7, 8, 10, 11	Analoge Input-Ports	Können (single ended) als 8 Eingänge oder (differentiell) als 4 Eingänge verwendet werden.
13, 14	Analoge Ausgabe	Zweimal. Details siehe Handbuch
21-28; 32-39	Digitale Ports	Können (laut Handbuch) einzeln als Eingabe oder als Ausgabe konfiguriert werden.

Tabelle 3.1: Pinbelegung der BreakoutBox [18]

Das “InstaCal”-Paket

Um das Gerät anzusprechen, wird Software von der Herstellerseite benötigt. Dort besteht die Auswahl zwischen verschiedenen Paketen. Empfehlenswert ist dabei das komplette

“InstaCal”-Paket. Das Paket ist kostenlos, daher fehlen einige Funktionen; diese sind allerdings für eine Basisfunktionalität nicht notwendig.

Kalibrierung

Laut Handbuch ist etwa alle sechs Monate eine Kalibrierung des Gerätes nötig, damit der analoge Eingangsport bei Messungen zuverlässige Werte liefert. Das Gerät kann sich selbst kalibrieren, im Verlauf der Kalibrierung müssen allerdings einige Klemmen z.B. mit Drahtbrücken verbunden werden. Die Software zeigt an den entsprechenden Zeitpunkten an, welche Drahtbrücken gerade benötigt werden.

Über das Programm *TracerDAQ* lassen sich Werte über die Zeit aufzeichnen, anzeigen lassen, eventuell speichern, usw.

Als kleines Beispiel wurde ein einzelner digitalen Eingabepin (Klemme 21) über einen Schalter abwechselnd mit 5V und GND verbunden. Dabei war zu ermitteln, wo diese Änderung zu sehen ist. Dies war auch mit folgenden Schritten und Einstellungen erfolgreich:

1. TracerDAQ starten
2. StripChart auswählen
3. Edit/DAQ Hardware Settings
4. Rechtsklick, Number of Channels: 1
5. DAQ-Channel: 1stPortA/Di0 (erster - und einziger - Digitalcontroller, PortA, Eingang 0; entspricht Pin 21)
6. OK-Knopf drücken
7. Play-Knopf drücken
8. Die Aufnahme beginnt und über den Schalter konnten zwischen den Pegeln gewechselt werden

Für einen komplexeren und praxisnahen Test wurde über die Software auch der Ultraschallsensor (siehe dazu auch: 2.1.2) ausprobiert:

Der Ultraschallsensor wurde wie folgt angeschlossen:

- SRF04-kompatibler Modus des Sensors (getrennte Trigger und Echo-Pins, siehe auch 2.1.2)
- GND des Sensors an Common Ground, 5V beispielsweise an den Spannungsausgang des PMD1208 (Klemme 30)
- “Echo”-Pin des Sensors an einen digitalen Eingangspin am PMD

- “Trigger”-Pin des Sensors an Pin 00 des PMD - jedoch als Eingang. Zum echten Triggern wurde der Schalter aus dem vorherigen Aufbau verwendet. Der Anschluss an den PMD diente nur zu einer besseren Verfolgung der Ereignisse am Bildschirm.

Grundsätzlich funktionierte dieser Testaufbau. Allerdings war in diesem Test zu erkennen, dass entweder das Gerät, die Software oder die USB-Verbindung dazwischen nicht schnell genug reagiert, um sehr kurze Impulse überhaupt zu erkennen.

In der Software-Suite TracerDAQ gibt es auch noch einen Oszilloskop-Modus. Dieser arbeitet zwar nur mit einem analogen Eingang zusammen, scheint aber im Timing erheblich besser; somit konnte der Echo-Impuls schon genauer ausgewertet werden.

Wenn die “Measurement Computing Universal Library” installiert wurde, ist der Zugriff auf die Breakoutbox auch über eigene Programme (zum Beispiel C) möglich. Wie für ein C-Programm üblich, muss die entsprechende Header-Datei (`cbw.h`) per `include` eingebunden werden und die entsprechende Library-Datei (`cbw.lib`) im Library-Pfad angegeben sein.

Auslesen eines Eingangs

Auch hier wurde mit einem einfachen Testaufbau begonnen. Wie schon im ersten Experiment wurde ein Schalter angeschlossen, mit dem der erste digitale Eingangspin (Klemme 21) zwischen GND und P5V umgeschaltet werden konnte. Das entsprechende Programmbeispiel ist Listing B.1 im Anhang.

Ansteuern eines Ausgangs

Das nächste Experiment war mit den gegebenen Programmen der Software-Suite nicht möglich: Per Software sollte ein Ausgang (dargestellt durch eine Leuchtdiode) abwechselnd an- und ausgeschaltet werden. Das entsprechende Programmbeispiel ist Listing B.2 im Anhang. Der dazugehörige Testaufbau war hier ebenfalls wieder sehr einfach:

- Die Kathode der Leuchtdiode wurde an GND angeschlossen
- Die Anode der Leuchtdiode wurde mit einen Vorwiderstand verbunden (etwa $1.5k\Omega$ für Low-Current-LEDs, etwa 150Ω für reguläre LEDs)
- Die andere Seite des Vorwiderstandes war mit der BreakoutBox verbunden. Für das Experiment wurde wieder Klemme 21 verwendet.

Erklärungen:

- `cbDConfigPort` unterscheidet sich in dem Punkt vom vorherigen Beispiel, dass der Port diesmal als `DIGITALOUT` geschaltet wird.
- `myResult` wird jeden Durchgang umgeschaltet.
- `cbDBitOut` unterscheidet sich kaum von `cbDBitIn`, allerdings kann hier der Parameter direkt übergeben werden. Die Verwendung eines Pointers ist nicht nötig.

Die weiteren Pins sind dann über 1-15 adressierbar. Dabei ist darauf zu achten, dass zwar weiterhin `FIRSTPORTA` in `cbDBitOut` auftaucht, aber in `cbDConfigPort` der zweite Port `FIRSTPORTB` angegeben werden muss. Das dritte Programmbeispiel geht daher nochmal auf dieses Problem ein.

Ein- und Ausgabe

Das nächste Programmbeispiel sollte nun die digitale Eingabe (des Schalters) in eine digitale Ausgabe (an der LED) umwandeln.

An dieser Stelle ließ sich dann ein Makel an der Programmbibliothek feststellen: Es ist nicht möglich, die digitalen Pins wirklich einzeln auf Ein- und Ausgang zu schalten. Trotz anders lautender Hinweise im Handbuch unterstützt die Bibliothek nur das Umschalten kompletter Ports. Für dieses Experiment wurde daher `FIRSTPORTA` für die Eingabe und `FIRSTPORTB` für die Ausgabe verwendet. Als Alternative könnte auch eine schnelle Umschaltung zwischen den Modi versucht werden.

Wie schon oben angegeben, funktioniert es nicht, `cbDBitOut` zur Ansteuerung des zweiten Ports `FIRSTPORTB` als Parameter zu übergeben. Allerdings ist es sehr wohl möglich, direkt alle acht Bits eines Ports in einem Zug zu setzen. Dazu dient dann die Funktion `cbDOut`.

Der Quelltext befindet sich im Anhang unter Listing B.3. Der Testaufbau ist prinzipiell eine Kombination aus den vorherigen beiden:

- Der Schalter schaltet wieder Klemme 21 (`FIRSTPORTA, 0`) zwischen P5V (Klemme 30) und GND (z.B. Klemme 40) um.
- Die Leuchtdiode wurde auf Klemme 32 (`FIRSTPORTB, 0` bzw. `FIRSTPORTA, 8`) gelegt, Kathode weiterhin an GND (z.B. Klemme 40)

Ultraschallsensor auslesen

Als letztes Experiment wurde versucht, den Ultraschallsensor automatisch auszulesen. Dabei bestand zunächst die Hoffnung, dass die eigene Programmierung unter C schneller ist als die Verwendung der mitgelieferten Software. Leider war dabei festzustellen, dass auch weiterhin die Latenzzeiten eine vernünftige Auswertung unmöglich machen und die von Windows bereit gestellten Funktionen an vielen Stellen zu ungenau sind. Messungen sind allenfalls bei größeren Abständen möglich. Der Vollständigkeit halber ist das Experiment hier dennoch dokumentiert.

Der Quelltext befindet sich als Listing B.4 im Anhang. Der Aufbau ergibt sich auch quasi von selbst:

- P5V des Sensors wird vom PMD-Gerät (Klemme 30) versorgt.
- GND des Sensors wird an einen der GND-Pins gelegt (z.B. Klemme 40)
- Der "Echo"-Ausgang des Sensors wurde an unserer BreakoutBox mit Klemme 21 (`FIRSTPORTA, 0`) verbunden. Damit muss dieser Port auch als Eingang beschaltet werden.

- Der “Trigger”-Eingang des Sensors wird an der BreakoutBox PMD-Gerät von Klemme 32 (FIRSTPORTA.8 bzw. FIRSTPORTB.0) ausgelöst.

Die fehlende Genauigkeit dieses Programms folgt aus der Ungenauigkeit der Methode `clock` (Die Millisekunden-Auflösung entspricht Differenzen von 17 cm) und der Latenz der USB-Aufrufe (etwa 10 Millisekunden pro Aufruf). In den Testreihen wurden immer nur drei verschiedene Ergebnisse (10,20 und 21 Millisekunden) geliefert.

Zudem sollte noch bemerkt werden, dass das Programm in dieser Form ohnehin nicht direkt auf dem Mikrocontroller laufen würde (Verwendung von Windows-Bibliotheken).

Da an dieser Stelle die Breakout-Box in Sachen Flexibilität und Geschwindigkeit nicht mehr ausreichte, wurde in späteren Experimenten hauptsächlich ein Mikrocontroller verwendet. Dies bot zudem den Vorteil, dass für Testzwecke geschriebener Quelltext relativ einfach in das endgültige Programm umgewandelt werden konnte.

3.1.3 Motortreiber

Im Folgenden wird auf die Notwendigkeit einer Motorsteuerung allgemein eingegangen, es werden zwei Standardbausteine betrachtet, bevor schließlich die Auswahl des Treibers begründet wird.

Im praktischen Einsatz muss sich der Roboter sowohl vorwärt als auch rückwärt fortbewegen. Da der Motor mehr Strom benötigt als ein Mikrocontrollerausgang liefern kann, muss die Ansteuerung über einen Motortreiber erfolgen. Es ist also eine Zusatzschaltung zur Trennung notwendig, sonst würde der Mikrocontroller unter der Last zerstört werden.

Standardbausteine

Eine zahlreiche Auswahl an Motortreiber ist von verschiedenen Chipherstellern als IC verfügbar. Um mit einem Mikrocontroller Digitalmotoren anzusteuern, kann der beliebten Motortreiber-ICs L293D (siehe Abb. 3.6) oder, für stärkere Motoren, der L298N (siehe Abb. 3.9) verwendet werden. Beide unterscheiden sich im zulässigen Strom, in der Gehäuseform und dass der L298 im Gegensatz zum L293D keine Freilaufdioden integriert hat.

L293D

Jedes IC des L293D beinhaltet zwei Brücken mit einer Belastbarkeit von jeweils 1200 mA. Die internen Schutzdioden an den jeweiligen Ausgängen begrenzen mögliche Spannungsspitzen. Das IC ist ohne Extramaßnahmen kurzschlussfest und besitzt einen internen Übertemperaturschutz, der die Ausgänge bei Überlastung abschaltet bis das IC ausreichend abgekühlt ist. Im Projekt ist der Motortreiber L293D eingebaut, wegen des internen Überhitzungsschutz. Dieser schützt den Motortreiber vor einer zu hohen Hitzeentwicklung. Die Motor-Stromaufnahme des RB 35 (Getriebemotor) beträgt unter Last ca. 300 mA. Der Motortreiber L298 ist für diesen Motortyp ungeeignet. Zudem müssten noch Freilaufdioden eingebaut werden, da die Motoren nicht direkt an die beiden Ausgänge des L298 angeschlossen werden (ohne Freilaufdioden) sollte. Zusätzlich schützen die Freilaufdioden den

IC vor den vom Motor ausgehenden Spannungsspitzen (siehe Abb. 3.9). Der Schaltkreis L298 beinhaltet ebenfalls zwei komplette H-Brücken mit der zwei Motoren angesteuert werden können. Die Pinbelegung ist ebenfalls dem L293D sehr ähnlich, jedoch verfügt er über eine andere Bauform. Der wichtigste Unterschied besteht darin, dass jede H-Brücke bei L298 bis zu 2A belastet werden kann. Es lassen sich also wesentlich größere Motoren ansteuern. Ein weiterer Vorzug sind die sogenannten SENSE-Ausgänge, über die der komplette Strom fließt. Mit dem L298N ist es also möglich, den Strom zu kontrollieren, den die Motoren ziehen. Dazu kann man die beiden CURRENT SENSING-Pins benutzen. Diese können genutzt werden, um die Belastung der Motoren im Einsatz zu überwachen und bei Überlastung über die Software einzugreifen.

Testschaltung

Es wurde eine Testschaltung gebaut, dazu genügte eine kleine SMD Platine, auf der der L293D Baustein befestigt wurde. Zuvor wurde mit dem Programm Eagle ein Schaltplan der Motorsteuerung erstellt (siehe Abb. A.5) aus der, dann das Board (siehe Abb. A.11) erzeugt wurde. Bei dem Test stellte sich heraus, dass der Motortreiber sehr robust ist und der Motortreiber unter den gegebenen Anforderung nicht zu stark erwärmt wird.

C++ Code zum Testen des Motors über eine Breakout-Box

Beim Testen der Schaltung wurde ein Testprogramm in C++ Code geschrieben, das die Motorsteuerung, wie in der Tabelle abgebildet (siehe Abb. 3.7), umsetzen soll. Dieses Testprogramm dient nur zum Testen der Schaltung. Der spätere Code, der in Atmel programmiert wird, wird sich von dem hier gezeigten Code sehr stark unterscheiden. Beim Testen der Schaltung hat sich herausgestellt, dass je kleiner die Periodendauer ist (siehe Abb. 3.8) , desto ruckartiger sich der Motor dreht. Beim Wechsel von High auf Low bzw. Low auf High ist der Motor kurzfristig in Zustand 1 (siehe Abb. 3.7). Dieser Code konnte aus zeitlichen Gründen nicht weiter verbessert werden. Die Abbildung 3.8 zeigt den Zusammenhang der Variable Periodendauer im C++ Programmcode. In der Abbildung sind drei unterschiedliche Zustände zu sehen (10 Prozent, 60 Prozent, 90 Prozent), die die Drehzahl des Motors beeinflussen. Je kleiner der Wert ist (in Prozent), desto langsamer ist die Drehzahl des Motors. Im Anhang befinden sich die entsprechende Header-Datei B.19 und das Main-Programm B.20.

3.2 ATMega

Wie weiter oben schon angeführt, wird im Roboter der Mikrocontroller ATMega32 von Atmel (Abbildung 3.10) verwendet. Dieser soll an dieser Stelle etwas genauer betrachtet werden. Da seine technischen Details zu ausführlich zum Auflisten sind, wird eher auf die Bereiche eingegangen, die für den Raptor relevant sind. Um sich länger mit dem Mikrocontroller zu beschäftigen, ist ohnehin eine umfangreiche Lektüre des Datenblattes und der Anleitungen im Netz unumgänglich.

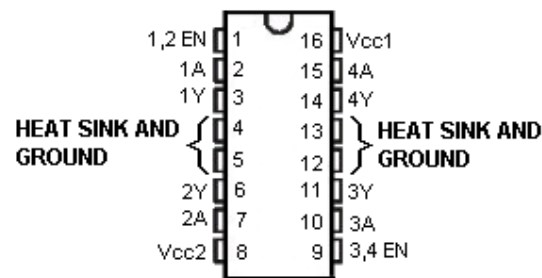


Abbildung 3.5: Der L293D Baustein [8]

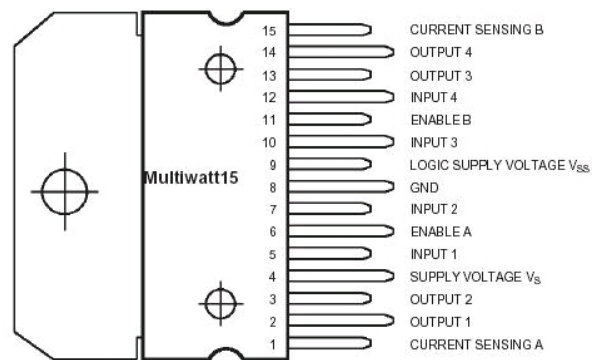


Abbildung 3.6: Der L298 Baustein [8]

Zustand	Motoreingänge		Ausgabe
	A	A'	
1	+	+	kurzschluss
2	+	-	vorwärt
3	-	+	rückwärts
4	-	-	weichebremse

Abbildung 3.7: Motoransteuerungstabelle [8]

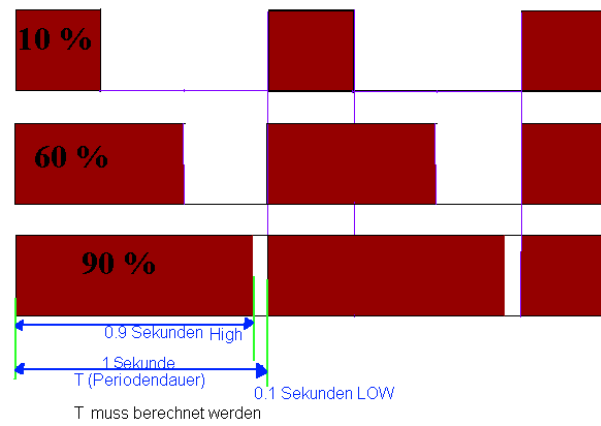


Abbildung 3.8: Bestimmung der Periodendauer [8]

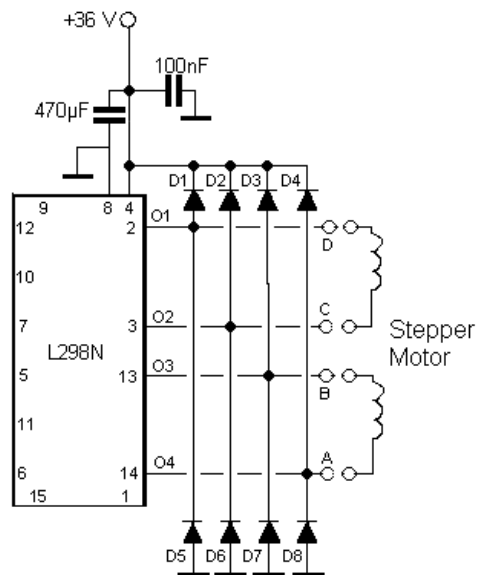


Abbildung 3.9: L298 mit Freilaufdioden [8]

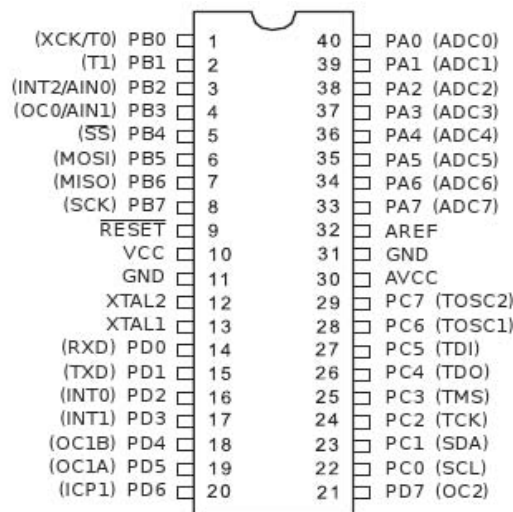


Abbildung 3.10: Pinbelegung des ATmega32

3.2.1 Programmierhardware

Auch wenn die Programmierung des Mikrocontrollers *in system* erfolgen kann, wird dazu Programmierhardware benötigt. Es gibt auf dem Markt eine Auswahl an Geräten und es existieren ausführliche Anleitungen zum Selbstbau. Im einfachsten Fall wird der Mikrocontroller quasi direkt an den Parallelport des PCs angeschlossen - was nicht empfohlen wird, da im Fehlerfall der Port beschädigt werden kann. Sicherer ist die Verwendung von Dioden oder einem kompletten Bustreiber[22]. Zudem gibt es Programmierhardware zum Anschluss an den seriellen Port des PCs ("Serial-ISP"). Aufgrund der großen Verbreitung von USB-Anschlüssen und des Komfort dieses Busses gibt es auch Adapter, um Mikrocontroller über den USB-Port zu programmieren.

AVR ISP mkII

Für die PG wurde ein *AVRISP mkII* (Abbildung 3.11) angeschafft. Dabei handelt es sich um einen USB-Programmer direkt von der Firma Atmel. Die Verwendung eines bereits montierten Programmers bietet den Vorteil, dass der Programmer auch direkt von der firmeneigenen Software-Suite unterstützt wird und im Gegensatz zu Eigenbau-Lösungen davon ausgegangen werden kann, dass er funktioniert - eine Fehlersuche dort kann entfallen.

Anschluss am Mikrocontroller

Auch wenn es verschiedenste Programmieradapter mit verschiedensten Anschlüssen auf der PC-Seite gibt (klassisch seriell, parallel oder USB), ist der Anschluss auf der anderen Seite standardisiert. Zwar gibt es auch hier noch verschiedene Bauformen:

- Die Firma Atmel definiert eine Belegung an einer sechspoligen Buchsenleiste. Diese findet sich auch am AVRISP mkII.
- In vielen Plänen im Netz wird von einer zehnpoligen Buchsenleiste ausgegangen. Für

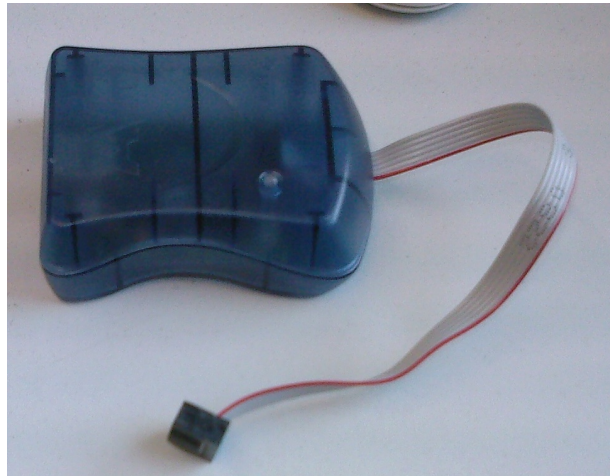


Abbildung 3.11: AVRISP mkII - unser Programmiergerät

diese lässt sich einfacher Anschlusshardware finden. Daher hat sich dieser Anschluss als ein “Standard der Hobbybastler” etabliert.

- In manchen Anwendungen wird ein neunpoliger D-Sub-Stecker (D shaped subminiature) verwendet - oftmals dient er zur Übergabe auf eine zehnpolige Buchsenleiste.

Im Wesentlichen sind dort aber immer die gleichen Signale enthalten. Tabelle 3.2 listet die verschiedenen Pinbelegungen sowie den Anschluss am ATMega und die Verbindungen zum Bustreiber-IC bei einem selbst gebauten Programmer mit Bustreiber[22] auf. Die Details werden später noch erläutert.

<i>Belegung</i>	<i>ATMega</i>	<i>6 pol Wanne</i>	<i>10 pol Wanne</i>	<i>9 pol DSub</i>	<i>IC</i>
MISO	7	1	9	5	11
VCC	z.B. 10	2	2	6	20, abgesichert
SCK	8	3	7	4	12/14
MOSI	6	4	1	1	18
Reset	9	5	5	3	3/5/7
GND	z.B. 11	6	4/6/8/10	7/8/9	10
LED	n/a	n/a	3	2	16

Tabelle 3.2: Die Pinbelegung verschiedener Stecker

Um die verschiedenen Adern eines Kabels an einer Buchsenleiste zu identifizieren, muss der Stecker so gehalten werden, dass die markierte Ader nach links zeigt. Pin 1 befindet sich dann oben links, die anderen Pins folgen im Zickzackmuster.(Abbildung 3.12)

Da der Programmieradapter nicht direkt an den Mikrocontroller angeschlossen werden kann und das verwendete Steckbrett keine direkte Anschlussmöglichkeit liefert (Es ergäbe sich immer eine Verbindung zwischen Signalen, die nicht verbunden werden dürfen), wurde

01	03	05	07	09	01	03	05
02	04	06	08	10	02	04	06

Abbildung 3.12: Pinbelegung von Wannenbuchsen

ein kleines Adapterboard (Abbildung 3.13) gebaut. Zum einen spreizt es die beiden Zeilen weit genug, dass diese auf getrennten Teilen des Steckbretts liegen, zum anderen können dort auch direkt Kabel eingesteckt werden. Eine Seite des Adapterboards ist markiert, dort muss die markierte Ader des Kabels (und somit Pin 1) sein.

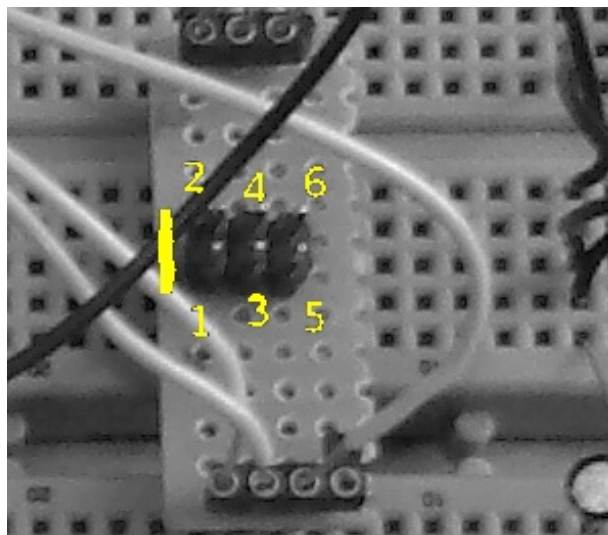


Abbildung 3.13: Das kleine Adapterboard für unseren Programmieradapter

3.2.2 Inbetriebnahme des ATmega32

Um den Mikrocontroller zum Laufen zu bringen, reicht es aus, die GND-Pins (Pin 11 und Pin 31) mit GND zu verbinden und den Spannungspin (Pin 10) an die Stromversorgung anzuschließen. Dennoch sollten weitere Verbindungen geschaffen werden:

- Zur Spannungsstabilisierung wird ein Kondensator (z.B. 100 nF) zwischen GND und P5V geschaltet. Solche Filter werden sehr häufig an ICs eingesetzt.

- Auch wenn der Mikrocontroller nicht für analoge Messungen verwendet wird, sollte auch AVCC (Pin 30) und AREF (Pin 32) mit P5V verbunden werden.
- In diesem Aufbau ist der Reset-Pin (Pin 9) unbeschaltet. Es kann also passieren, dass sich der Controller ständig zurücksetzt. Um den Pin auf einem sicheren Pegel zu halten, sollte er mit einem PullUp-Widerstand (z.B. $10\text{ k}\Omega$) auf P5V gezogen werden. Der Controller setzt sich zurück, wenn er einen GND-Pegel am Reset-Pin entdeckt. Somit ist P5V der “sichere” Pegel.

An diesem Punkt hat der Controller noch keinerlei Möglichkeiten, sich mit der Außenwelt zu verständigen (LEDs, Datenverbindungen, etc. . .). Eventuell zeigt ein Ampèremeter einen Stromverbrauch an, aber im Grunde ist nur zu hoffen, dass er auch läuft.

Erste Kommunikation mit dem PC

Die zuverlässigste Methode, den Mikrocontroller am PC zu erkennen, ist, das Programmiergerät anzuschließen. Die grundsätzlichen Verbindungen lassen sich aus Tabelle 3.2 entnehmen. Dabei ist allerdings zu beachten:

- Der PullUp-Widerstand am Reset-Pin wird nicht entfernt. Der Programmierer wird parallel dazu geschaltet. Generell sollte der Widerstand während aller Experimente an seinem Platz belassen werden.
- Weder kann das Programmiergerät den Mikrocontroller mit Strom versorgen, noch umgekehrt. Beide Bauteile müssen von einer externen Spannungsquelle gespeist werden. Zwar sind mit dem AVRISP mkII einige Basisoperationen schon durch die 5 Volt des USB-Ports möglich, aber für die echte Verwendung ist die Speisung des Controllers eine Notwendigkeit.
- Allen Bauteile in den Aufbauten sollten den gleichen Massepegel haben. Auch das Programmiergerät sollte keine Ausnahme sein.

Programmiersoftware

Auf der Homepage des Herstellers[2] lässt sich kostenlos das “AVR Studio” herunterladen. Dieses kann dann verwendet werden, um den USB-Programmer anzusprechen. Richtig interessant wird das Studio allerdings erst, wenn zusätzlich noch “WinAVR” [32] installiert wird; dieser C-Compiler wird automatisch in das AVR Studio integriert und C-Programme können von dort aus kompiliert und auf den Mikrocontroller übertragen werden.

Nach Start des Studios lässt sich über

Tools \ Program AVR \ Connect

der Programmer (AVRISP mkII) auswählen. In den verschiedenen Menüs sollte dann - wenn nichts falsch gemacht wurde - die Verbindung zum Controller zu erkennen sein. Dieser Dialog kann auch zum Aufspielen von kompilierten Programmen verwendet werden.

Selbstverständlich können damit auch Programme aufgespielt werden, die nicht mit AVR Studio kompiliert wurden. Beispielsweise gibt es auch Pascal, Fortran, Python, Basic und sogar Java-Compiler, die Maschinencode für den Mikrocontroller erzeugen.

Die ersten Testprogramme wurden in Basic geschrieben, daher folgen hier einige Programmbeispiele in Basic.

3.2.3 Programmierung in Basic

Der verwendete Compiler “BasCom”[3] ist zwar keine freie Software, es gibt jedoch eine kostenlose Demo-Version, die für die ersten Experimente völlig ausreichte.

Das erste Programm

Zugegeben, Listing B.5 ist ein Nullprogramm, allerdings ist es nicht wirklich trivial, es in den Controller zu übertragen. Es ist sicherlich besser, die ersten Fehler hier zu machen, als sie später in echten Programmen zu suchen.

In der Tat gab es schon hierbei die ersten Hindernisse. Wenn die Programmierung mittels AVR Studio nicht funktioniert, kann es notwendig sein, verschiedene Programmiergeschwindigkeiten - zu finden unter “Board” bei “ISP Freq.” - auszuprobieren. Oftmals reichte es auch aus, die Programmierung einfach noch zwei oder dreimal zu versuchen, bis das Programm dann doch erfolgreich in den Speicher des Atmels geschrieben wurde. In späteren Versuchen war es jedoch notwendig, vor der Programmierung die Taktfrequenz per “Fuse” herabzusetzen und nach der Programmierung wieder auf den “richtigen” Wert zu stellen. Erklärungen zu den benötigten Fuse-Bits finden sich in Kapitel 3.2.4

Zum Programm:

Zeilen 1ff Die Zeilen, die mit \$ beginnen, sind Direktiven an den Compiler. Beispielsweise wird hier der richtige Controller ausgewählt(`m32def.dat` enthält die Definitionen für den Mega32), Programmkonstanten festgelegt und per `$crystal` wird die Taktfrequenz in Hz eingestellt.

Zeilen 7f Das Hauptprogramm besteht dann aus einer einfachen Endlosschleife. Eine solche Schleife findet sich in den meisten Programmen und unterschiedlichsten Programmiersprachen. Mikrocontrollersoftware terminiert nicht; sie läuft in einer Endlosschleife.

Zeile 10 Somit ist das End eigentlich unsinnig. Aus Gründen einer übersichtlicheren Programmstruktur (und der Genauigkeit von Compilern) sollte man aber nicht darauf verzichten.

Ansteuerung einer LED

Zunächst einmal sollte der Mikrocontroller die BreakoutBox in den Testaufbauten ersetzen. Daher waren die ersten “echten” Programme dann auch wieder nur dazu gedacht, Taster auszulesen, Leuchtdioden anzusprechen und eine Kommunikation zum PC zu schaffen.

Listing B.6 bringt eine an Pin 21 (*Portc.0*) angeschlossene Leuchtdiode zum Blinken. Hierbei ist zu beachten, dass die Speisung der LED sicherheitshalber von der Stromquelle und nicht vom ATmega32 übernommen werden sollte. Somit wird diesmal die Kathode der LED an den Controllerpin angeschlossen und die Anode an P5V. Selbstverständlich muss weiterhin der Vorwiderstand im Stromkreis zu finden sein, seine Position ist aber beliebig. Das heißt natürlich dann, dass bei

```
Portc.0=1
```

ausgeschaltet wird. Sie hat dann zwischen den P5V am Controllerbeinchen und den P5V der Spannungsquelle kein Leuchtpotenzial.

Programm mit Taster

Neben der Ansteuerung von Ausgängen müssen natürlich auch Sensordaten verarbeitet werden, also die Eingänge ausgelesen werden können. An diesem Punkt der Experimentreihe bestand allerdings noch keine Datenverbindung zum PC, daher wurde der Tasterstatus über die Leuchtdiode anzeigen. Listing B.7 lässt eine Leuchtdiode weiterhin blinken, wechselt allerdings die Frequenz, wenn ein an Pin 33 (*PortA.7*) und GND angeschlossener Taster gedrückt wird.

Unbeschaltete Eingänge

Wenn der Taster gedrückt ist, ist der Eingang mit GND verbunden, kann in Programmen also als LOW ausgelesen werden. Ist der Taster allerdings nicht gedrückt, so gilt der Eingang als unbeschaltet, hat also keinen definierten Pegel.

Da in diesem Fall dennoch ein definierter Pegel gewünscht ist, werden in solch einem Falle oft *Pull-Up-Widerstände* eingesetzt, um unbeschaltete Eingänge auf HIGH zu ziehen.

Der ATmega32 stellt solche Pull-Up-Widerstände an den Eingängen auf Wunsch selbst zur Verfügung. Laut einigen Berichten sind diese zwar nicht sehr vertrauenswürdig, für kleinere Anwendungen reichen sie jedoch völlig aus.

Pull-Up-Widerstände aktivieren

Die internen Pull-Up-Widerstände werden aktiviert, indem die Pins mit 1 beschrieben werden, obwohl sie als Eingang geschaltet sind. Natürlich können die Eingänge dann nicht mehr mit `Portx.y` angesprochen werden, zum Auslesen ist dann `Pinx.y` zuständig.

3.2.4 Einbindung eines Quarzes

Bis zu diesem Punkt läuft der Mikrocontroller nur mit 1MHz. Dieser Takt wird von einem internen Taktgenerator erzeugt. Um den ATmega32 mit voller Geschwindigkeit laufen zu lassen, sollte ein externer Quarz verwendet werden. Auf Hardware-Seite wird ein Quarz-Baustein zwischen die XTAL-Leitungen (Pin 12 und Pin 13) gesetzt, zusätzlich werden die beiden Leitungen über einem Kondensator (z.B. 22pF) mit GND verbunden.

Damit der Controller nun nicht mehr seinen internen, sondern diesen externen Taktgeber verwendet, muss ein sogenanntes *Fuse*-Bit umgesetzt werden. Das AVR Studio verfügt

über einen eigenen Dialog zum Setzen der Fuse-Bits, welcher über das Registertab “Fuses” ausgewählt werden kann, wenn die Verbindung zum Mikrocontroller hergestellt wurde (siehe auch Kapitel 3.2.2). Leider verfügt der Controller über eine Menge von Fusebits und Konfigurationsmöglichkeiten und auch das entsprechende Kapitel im Datenblatt ist keineswegs als trivial zu bezeichnen. Für einen einfachen schnellen Quarz (Wir haben 16MHz verwendet) funktionierte die Einstellung $CKSEL = 1111$ und $SUT = 01$ ganz gut.

Damit der Compiler nun auch diese höhere Taktrate mit in seine Berechnungen aufnehmen kann - zum Beispiel bei Warteschleifen, erwartet er die Angabe der korrekten Taktrate. Dies geschieht in BasCom über

```
$crystal = 16000000
```

Verbindung zum PC

In späteren Experimenten wurden doch mehr Debug-Ausgaben nötig, als eine Leuchtdiode bieten konnte. Auf die Kommunikationsverbindung konnte daher nicht mehr verzichtet werden.

Leider ist eine solche Ausgabe über den Programmieradapter nicht möglich. Der ATmega32 besitzt jedoch einen integrierten Baustein (USART) für serielle Kommunikation, kann also eher einfach mit dem COM-Port eines PCs verbunden werden.

Allerdings ist der direkte Anschluss nicht möglich. Zwar verwendet der ATmega auch das RS232-Protokoll, er verwendet dabei jedoch andere Pegelwerte:

RS232- und TTL-Pegel

Der serielle Port des PCs verwendet das RS232-Protokoll mit RS232-Pegeln:

- Ein logisches HIGH entspricht etwa -10V
- Ein logisches LOW entspricht etwa +10V

Auf der anderen Seite arbeitet der ATmega32 (wie auch die meisten anderen ICs) mit TTL-Pegeln (Transistor-Transistor-Logik)

- Ein logisches HIGH entspricht P5V
- Ein logisches LOW entspricht 0V

Damit ATmega und PC trotzdem miteinander kommunizieren können, muss ein Pegelwandler verwendet werden - Der *MAX232* ist ein Standardbaustein für diese Aufgabe.

Verdrahtung des MAX232

Im Testaufbau funktionierte nun folgende Verbindung:

- Der IC wird an die gleiche Stromversorgung wie der ATmega32 angeschlossen:
 - 15 GND
 - 16 P5V

- Zur Filterung sollte wieder ein 100nF-Kondensator zwischen diese beiden Anschluss-pins gesetzt werden.
- Zudem werden als Grundbeschaltung noch vier Kondensatoren benötigt. In verschiedenen Quellen (Datenblättern, Tutorials, Schaltplänen) findet man verschiedene Werte (z.B. $4,7\mu F$ oder $10\mu F$), der detaillierte Wert ist also nicht so relevant.
 - ein Kondensator zwischen Pin1 und Pin3, der Minus-Pol geht an Pin3
 - ein Kondensator zwischen Pin4 und Pin5, der Minus-Pol geht an Pin5
 - ein Kondensator zwischen Pin2 und GND, der Minus-Pol geht an GND
 - ein Kondensator zwischen Pin6 und GND, der Minus-Pol geht an Pin6
- Datenverkehr vom Atmel zum PC (Abbildung 3.14)

ATMEL TxD (Pin15) – > MAX232 T1In (Pin11)

- Am MAX232: Umwandlung in RS232-Pegel

MAX232 T1Out (Pin14) – > PC RxD (Pin2)

- Datenverkehr vom PC zum Atmel

PC TxD (Pin3) – > MAX232 R1In (Pin 13)

- Am MAX232; Umwandlung in TTL-Pegel

MAX232 R1Out (Pin12) – > ATMEL RxD (Pin14)

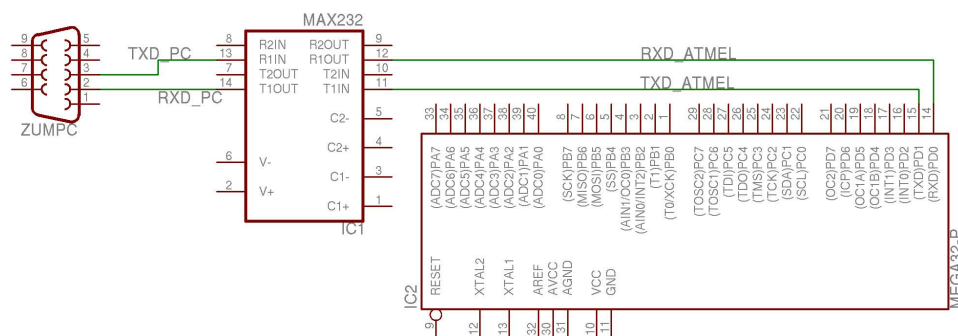


Abbildung 3.14: Verbindung zwischen ATMega und PC - nur Basis

Ausgabe im Basic-Programm

Bascom erwartet eine weitere \$-Direktive `baud`, anschließend funktioniert die Ausgabe unkompliziert über den `print`-Befehl. (Siehe Listing B.8)

Verbindung am PC

Am PC wird nun das bevorzugte Terminalprogramm verwendet (beispielsweise *minicom* unter Linux oder *HyperTerminal* unter Windows). Dabei ist zu beachten, dass eine Flusskontrolle mit Hardware nicht möglich ist.

Auslesen des Ultraschallsensors unter Basic

Nachdem die Experimente mit der BreakoutBox am Ultraschallsensor gescheitert sind, sollte der Mikrocontroller bessere Ergebnisse liefern. Das dazugehörige Listing (B.9) wurde fast 1:1 der Sensor-Dokumentation übernommen. Da sich die Details dieses Listings hauptsächlich auf den Basic-Compiler - der ja später nicht mehr verwendet wurde - oder auf den Ultraschallsensor - welcher auch bereits ausreichend dokumentiert wurde - beziehen würde, wurde hier auf weitere Details verzichtet.

3.2.5 Umstieg auf C

Später wurde der Mikrocontroller nicht mehr unter Basic programmiert. Grundsätzlich eignet sich C für hardwarenahen und schnellen Code besser als Basic und ist dennoch wartbarer als beispielsweise Assembler.

WinAVR

Mit der Installation von WinAVR stand schon eine vollwertige AVR-Toolchain zur Verfügung, auch war direkt die *avr-libc* vorhanden, die ebenfalls benötigt wird. Im Netz befinden sich zwar Anleitungen, wie Makefiles zum Umgang mit AVR-Mikrocontrollern erstellt und die Toolchain im Detail bedient werden, aber mit AVR Studio als graphischer Oberfläche reicht es, mit F7 das Programm zu übersetzen und über den Programmierdialog direkt auf den Controller zu überspielen. Wenn vorher in den Projektoptionen (*Project\Configuration Options*) unter *General* der Controller und die Taktfrequenz richtig eingestellt, so sind einige Konstanten schon richtig definiert, die sonst in ein Makefile oder direkt in den Quelltext eingefügt werden müssten.

Das Nullprogramm unter C

Wie auch beim ersten Programm unter Basic wird hier mit einem Nullprogramm begonnen, um die Verwendung des Compilers und die Rahmenbedingungen im C-Compiler auszutesten. (Listing B.10)

Bemerkungen:

- Die *main*-Methode hat - wie alle C-Methoden - einen Parameterblock (hier *void*). Natürlich können dem Mikrocontroller keine Parameter an die *main*-Methode übergeben werden, dennoch gehört diese Parameter-Definition zum C-Standard und muss daher vorhanden sein.
- Analog: Die *main*-Methode hat einen *int*-Rückgabewert, auch wenn dieser nirgendwo abgelesen werden kann. Ebenfalls verlangt der Compiler einen *return*-Aufruf, auch wenn dieser vom Programm nie erreicht werden darf.
- Wie auch bei Basic-Programmen sollte niemals ein Programmende erreicht werden, sondern immer mit einer Endlosschleife gearbeitet werden. Anderenfalls würde der *Program Counter* in Speicherbereiche laufen, die nicht beschrieben wurden, also undefiniert sind.

Ansteuern von Ausgängen

Auch unter C müssen die Ausgänge angesteuert werden, also wurde auch hier wieder eine Leuchtdiode angeschlossen und über ein Programm (Listing B.11) anzusteuern versucht.

Bemerkungen:

- Da zu dem Zeitpunkt noch keine Funktionen für Warteschleifen eingebunden waren, wird die Leuchtdiode einfach nur eingeschaltet - wie bisher wird dabei der Pin auf GND gezogen.
- Die Ansteuerung der Pins des Mikrocontrollers gehört zu den I/O-Funktionen, daher muss auch `avr/io.h` eingebunden werden.

Technisch gesehen wird in der `io.h` die richtige Header-Datei für den passenden Controller ausgewählt. Dieser wird dem Compiler über eine Konstante übergeben. Wie schon oben angemerkt, wird dies beim AVR Studio direkt über die Projekteinstellungen gemacht.

- Wie auch in Basic muss definiert werden, dass der Port als Ausgang verwendet wird. Dies geschieht über DDRC - das Data Direction Register für PortC. Dieses Register wird als komplettes Byte geschrieben.
- Anschließend kann dann der Pin gesetzt werden. Auch PORTC wird komplett byteweise gesetzt. Dabei entspricht 1 dem HIGH-Pegel.
- Nach Definition eines Mikrocontroller-Programmes gehört jeglicher Quelltext vor der Endlosschleife zur *Initialisierung*. So gesehen hat das Programm keine Funktion, die LED wird in der Initialisierung angesteuert.

Einlesen von Eingängen

Nachdem nun das Ansteuern von Leuchtdioden funktionierte, wurde als nächstes das Einlesen von Eingängen in Angriff genommen. Für das dazugehörige Programm (Listing B.12) wurde ein echter Umschalter an Pin 34 (PortA.6) angeschlossen. Um auf den PullUp-Widerstand zu verzichten, wurde dieser mit den Umschalt-Eingängen an GND und P5V angeschlossen.

Wie auch im Basic-Beispiel wird der ausgelesene Status des Schalters auf eine Leuchtdiode abgebildet. Die LED befindet sich weiterhin an Pin 31.

- In diesem Beispiel wird der komplette PortA zum Eingangsport und der komplette PortC zum Ausgangsport deklariert.
- Auf den PullUp-Widerstand wurde verzichtet. Ansonsten müsste dieser wie gewohnt mit PORTA aktiviert werden. Wie in Basic geschieht das Einlesen des Zustands am Controllerpin über ein anderes Schlüsselwort: `PIN`

- Bei der Abfrage `PINA & 0x40` wird aus dem gelesenen Status an Port A ein einzelnes Bit herausgefiltert. Diese Operation ist als *Bitmasking* bekannt. Äquivalent zu 0x40 (hexadezimal) sind 64 (dezimal), $(1 << 6)$ (Bitshifting) und 0b01000000 schreiben. Allerdings sind einige Varianten lesbarer als andere.

Serielle Übertragung

Auch unter C ist eine Ausgabe von Text gewünscht - insbesondere Debugging-Informationen sollten ablesbar sein.

Wie bisher wird dabei der UART des Mikrocontrollers verwendet, um an die serielle Schnittstelle des PCs Daten zu senden. Der bisherige Aufbau mit dem Pegelwandler *MAX232* kann beibehalten werden, allerdings wird die Datenübertragung erheblich komplizierter. Unter C werden die Register direkt angesprochen und für eine detaillierte Kenntnis dieser Register ist eine gründliche Lektüre des ausführlichen Datenblattes des Controllers nötig. Hier soll nun allerdings nicht auf die einzelnen Flags und ihre Funktionen eingegangen werden - die meisten Flags sind schon beim Start des Controllers auf die gebräuchlichste Konfiguration gesetzt -, statt dessen wird nur kurz der nötige Quelltext angegeben.

Die Methoden (Listing B.13) sollten im endgültigen Programm wahrscheinlich besser in eine eigene Datei ausgegliedert werden. Sie implementieren auch nur die serielle Ausgabe, das Thema der seriellen Eingabe wurde bisher nicht behandelt. Für Debug-Zwecke sollte die Ausgabe von Informationen allerdings ausreichen.

- Die Definition von `F_CPU` ist nicht nötig, da sie von AVR Studio automatisch eingefügt wird.
- Die gewünschte Baudzahl muss abhängig vom Prozessortakt noch umgerechnet werden. Die Formel für die Umrechnung soll hier nicht weiter dokumentiert werden.
- Die Aktivierung des Ports und die Einstellung der richtigen Geschwindigkeit erfolgt über `uart_init`, einzelne Zeichen lassen sich mit `uart_putc`, ganze Zeichenketten mit `uart_puts` ausgeben.
- Aufgrund der Warteschleife in Zeile 11 blockiert die Zeichenausgabe den gesamten Prozessor, bis der Text komplett ausgegeben ist. Das ist nicht effizient, sollte für Debug-Ausgaben aber kein Problem sein. In der Regel sollte lediglich eine Fehlermeldung vom Controller erfolgen, anschließend kann das Programm komplett zurückgesetzt werden.

Warteschleifen

Auch in C werden an vielen Stellen Wartezeiten benötigt - beispielsweise um Impulsen eine gewisse Länge zu garantieren. Die *avr-libc* bietet zwar Methoden, die ähnlich wie die `waitms`-Methode in Basic verwendet werden können. Hier sind allerdings noch Besonderheiten zu beachten.

In der `util/delay.h` befinden sich die Methoden:

```
_delay_ms(int);  
_delay_us(int);
```

Mit diesen ist es prinzipiell möglich, den Programmfluss für die gewünschte Anzahl an Millisekunden oder Mikrosekunden anzuhalten. (Intern werden so viele NOP-Operationen durchlaufen, wie für die Wartezeit notwendig ist) Allerdings unterliegen diese Methoden einer Beschränkung. Beispielsweise heißt es bei der Methode `_delay_ms_`:

The maximal possible delay is 262.14ms/F_CPU in MHz

Um auch größere Werte zu erlauben, wurde eine kleine Wrapper-Funktion geschrieben. (Listing B.14. Auch diese sollte in eine externe Datei ausgegliedert werden. Zu Beachten dabei ist, dass etwa alle 262ms durch die Operationen der Schleife zusätzliche Taktzyklus verwendet, aber nicht von der Wartezeit abgezogen werden. Außerdem wird - wie bei den UART-Methoden - die gesamte Rechenzeit mit Warten verbracht.

3.2.6 Interrupts

Im Raptor werden wir an verschiedenen Stellen Interrupts verwenden. Beispielsweise ist das Ereignis “Der Roboter ist gegen eine Wand gefahren” als so kritisch anzusehen, dass sofort die Radmotoren angehalten werden sollen. “Sofort” bedeutet dabei implizit “unabhängig davon, welche andere Aufgabe der Controller in dem Moment bearbeitet”. An diesem Beispiel soll nun dokumentiert werden, wie im ATmega32 mit Interrupts gearbeitet werden kann.

Allgemeines zu Interrupts

Interrupts dienen zur Kontrolle des Programmablaufes - genauer gesagt: Über sie kann eine kontrollierte Störung des Programmablaufes erreicht werden.

Wenn ein nicht-maskierter Interrupt ausgelöst wird, so wird der normale Programmablauf sofort unterbrochen und das Programm springt an die entsprechende Interrupt-routine. Nach Abarbeitung dieser Routine setzt das Programm die Arbeit dort fort, wo es unterbrochen wurde - wenn nicht zwischendurch eine neue “Unterbrechungsanforderung” eingetroffen ist.

Dieser Sprung aus dem Hauptprogramm heraus kann einige Probleme mit sich bringen. GOTO ist eigentlich immer ein sehr unschönes Programmierkonstrukt und bei einem Interrupt kann der Prozessor eventuell bei einer “wichtigen” Aufgabe unterbrochen werden. Dennoch können wir auf Interrupts nicht verzichten. Zudem lassen sich durch sie auch wieder einige sehr elegante Programme implementieren.

Da bei einem Interruptsprung der reguläre Programmablauf gestört wird, sollte zumindest versucht werden, diese Störung zu minimieren. Es ist eine allgemeine Empfehlung, in den Interruptroutinen nur möglichst wenig Programmaufwand unterzubringen und die echte Verarbeitung der Daten auszulagern.

Quellen eines Interrupts im ATmega32

Es gibt verschiedene Quellen, die einen Interrupt auslösen können - beispielsweise ein leerer Puffer, eingetroffene Daten vom USART, erreichte Schwellwerte bei Zählern oder der Überlauf eines Wertes. Sehr viele Funktionsblöcke im Mikrocontroller haben ein Flag "Interrupt Enable". Diese Interrupts sind ausführlich im kompletten technischen Datenblatt von Atmel dokumentiert und werden daher hier nicht detailliert erklärt.

Für den sofortigen Stopp der Motoren bei Kollision (erkannt durch einen "Bumper") wird ein "externer Interrupt" verwendet. Dieser wird ausgelöst, wenn an einem bestimmten Pin des Controllers "etwas passiert". Andere Interrupts werden auf ähnliche Weise aktiviert und "abgefangen".

Externer Interrupt

Wie schon erwähnt, lassen sich die meisten der Port-Pins am ATmega32 auf besondere Funktionen umschalten. Der externe Interrupt ist eine solche Funktion; für INT0 (den ersten externen Interrupt) ist Pin 16 vorgesehen. In dem Moment, in dem dieser als externer Interrupt aktiviert wird, wird PortD.2 von der "normalen" I/O-Kontrolle abgekoppelt.

Dennoch muss der Pin als Eingang beschaltet werden. Es ist auch eine Beschaltung als Ausgang möglich; hierüber lassen sich Interrupts implementieren, die von Software-Seite aus durch schreibende Ansteuerung des Pins ausgelöst werden. Hierbei muss gegebenenfalls der interne Pull-Up-Widerstand aktiviert werden.

```
DDRD  |= (1<<PD2);  
// PORTD |= (1<<PD2);
```

Für die Bumper am RAPTOR ist der interne Vorwiderstand nicht nötig, da eine externe Beschaltung erfolgt, die den Pegel auf HIGH lässt, wenn kein Taster gedrückt ist.

Bisher wurde nur gesagt, dass ein Interrupt ausgelöst wird, wenn an diesem Pin "etwas passiert". Allerdings gibt es verschiedene, einstellbare Vorraussetzungen für das Erzeugen des Interrupts:

- LOW-Pegel
- fallende Flanke
- steigende Flanke

Da ein Bumper den Pegel auf LOW zieht, muss der Interrupt auf "fallende Flanke" ausgelöst werden. Der richtige Konfigurationsbefehl dazu lautet:

```
MCUSR |= (1<<ISC01);
```

Die anderen Fälle werden im Datenblatt genauer dokumentiert.

Der eigentliche Interrupt wird dann über das Global Interrupt Control Register aktiviert.

```
GICR |= (1<<INT0)
```

Dennoch werden an diesem Punkt noch keine Interrupts erzeugt. Damit der Atmel überhaupt mit Interrupts arbeitet, muss diese Funktionalität global über das “Global Interrupt Enable”-Flag aktiviert werden. Dies geschieht diesmal nicht direkt durch Beschreiben eines Registers, sondern über den Befehl `sei()`; . Diese Methode ist ein Bestandteil der AVR-Interrupt-Bibliothek, welche gesondert eingebunden werden muss.

```
#include <avr/interrupt.h>
```

Der Gegenbefehl zum globalen Deaktivieren von Interrupts lautet `cli()`;

Ab diesem Punkt reagiert der ATmega auf diesen Interrupt. Natürlich müssen wir ihm noch sagen, wie er auf den Interrupt reagieren soll. Dazu schreiben wir die Interruptbehandlungsroutine:

```
ISR(INT0_vect) {  
    motor_aus();  
    // uart_puts("IRC!\n\r");  
}
```

- `ISR` ist das Schlüsselwort für Interruptroutinen
- `INT0_vect` ist das Schlüsselwort für exakt diesen Interrupt (Der Name findet sich in der entsprechenden .h-Datei oder in der Dokumentation von Atmel).
- Mechanische Taster prellen. Somit wird die Interruptsbehandlungsroutine meist sehr schnell mehrfach hintereinander aufgerufen. Da sie in diesem Falle lediglich Motoren ausschaltet, die zu dem Zeitpunkt bereits ausgeschaltet sind, erschien eine Entprell-Routine nicht wirklich sinnvoll.

3.2.7 I²C auf dem Atmel

Die ATmega-Baureihe kann direkt an einen *I²C*-Bus angeschlossen werden (*I²C* heißt bei Atmel *Two Wire Interface*). Als ein Test wurde der Kompass-Sensor vom Atmel ausgelesen (Da diese Aufgabe vom Gumstix übernommen wird, wurde dies hier nicht dokumentiert) und hatten auch einen Testaufbau, in dem ein ATmega32 als Master einem anderen ATmega32 als Slave Daten zugeschickt hat. Da das dort verwendete Programm auch mit einem auf Slave geschalteten ATmega32 und dem als Master verwendeten Gumstix funktioniert hat, könnte es die Grundlage für den späteren Datenaustausch zwischen Gumstix und ATmega werden.

Das im Anhang abgebildete Listing B.15 wurde leicht gekürzt. Insbesondere wurden schon in vorherigen Kapiteln erwähnte Methoden nicht erneut definiert. Die *TWIerror*-Methode dient nur dazu, Statuswerte, die nicht erwartet wurden, anzuzeigen und den Programmablauf anschließend zu stoppen. Der *TWI*-Interrupt wird nicht verwendet und das Programm ist nicht fehlerrobust. Wie schon angemerkt, soll es auch erstmal nur ein Beispiel sein.

Timer/Counter des ATMegas

Der ATmega32 hat verschiedene eingebaute Timer/Counter. Diese können beispielsweise für eine Pulsweiten-Modulation verwendet und zur Ansteuerung eines Motortreibers genutzt werden. In diesem Beispiel soll dokumentiert werden, wie Timer/Counter0 zur Impulsmessung am Ultraschallsensor genutzt werden kann. Um Pins einzusparen, wurde hierbei schon der gemischte 1-Pin-Modus (siehe 2.1.2) des Ultraschallsensors verwendet. Dabei sollte außerdem das Umschalten der Datenflusskontrollregister mitten im Programmablauf dokumentiert werden.

Das Programm befindet sich im Anhang unter Listing B.16. Wie auch das TWI-Listing soll es nur eine Grundlage bilden. Die Funktionen sollten später noch in andere Dateien ausgelagert werden.

Angeschlossen wurde der gemeinsame Pin des Sensors an Pin01 (PortB.0) des Mikrocontrollers. Diese Auswahl war rein willkürlich, ein besonderer Pin ist für die Messung nicht notwendig.

Der Counter kann verschiedene Vorteiler verwenden, um nicht bei jedem Taktzyklus zu zählen. Leider lassen sich selbst bei größtem Vorteiler Overflows nicht vermeiden. In einer Overflow-Interruptroutine werden diese Overflows gezählt.

3.3 Herstellung von Platinen

Einfache Leiterplatten bestehen aus einem elektrisch isolierenden Trägermaterial, dem sogenannten Basismaterial, auf dem eine oder zwei Kupferschichten aufgebracht sind. Die typische Schichtstärke beträgt $35\ \mu m$, für Anwendungen mit höheren Strömen werden Stärken zwischen $70\ \mu m$ und $140\ \mu m$ verwendet. Als Basismaterial werden in der Projektgruppe RAPTOR mit Epoxidharz getränkte Glasfasermatten verwendet. Auf die Kupferschicht ist Fotopositivlack aufgetragen, der mit einer lichtundurchlässigen Schutzfolie abgeklebt ist. Bei den zweiseitigen Platinen ist die Kupfer-, Fotolack- und Schutzschicht auf beiden Seiten. Die Platinen für den Staubsaugerroboter ist in sieben Schritten manuell von den Mitgliedern der Hardwaregruppe hergestellt worden. Die verwendeten Methoden werden im Folgenden näher beschrieben. Dabei wird auf die gewonnenen Erfahrungen und die eingesetzten Geräte sowie Materialien näher eingegangen.

3.3.1 Layout

Zum Erstellen des Layouts der Platinen wurde das Programm EAGLE (Einfach Anzuwendender Grafischer Layout-Editor) der Firma CADSoft [9] benutzt. Die Software besteht aus mehreren Komponenten: Layout-Editor, Schaltplan-Editor, Autorouter und einer erweiterbaren Bauelementedatenbank. Einige Bibliotheken, wie beispielsweise der Baustein CNY70 (Lichtsensoren), mussten hinzugefügt werden. Der Autorouter erzielt nicht immer das gewünschte Ergebnis, so dass manuelle Verbesserungen vorgenommen werden müssen. Zusätzlich ist darauf zu achten, dass die Leiterbahnbreiten nicht zu klein, die Abstände

zwischen verschiedenen Leiterbahnbreiten groß genug und eine ausreichende Lochgröße der Pads vorhanden ist.

3.3.2 Belichtung

Das Layout wird zunächst auf eine transparente Folie gedruckt. Danach kann die Platine, ohne Abziehen der Schutzfolie, auf die richtige Größe zugeschnitten werden. Mit der bedruckten Seite nach unten kann das Layout auf die Platine, inzwischen ohne Schutzfolie, gelegt werden. Bei zweiseitigen Platinen werden die beiden Layoutfolien zuerst übereinander gelegt und anschliessend mit Tesafilm an den Seiten fixiert. Die zurechtgeschnittene Platine kann zwischen die Folien geschoben werden. Das Layout und die Platine wird auf die Glasplatte des Belichtungsgerätes gelegt. Um mögliche Fehlerursachen zu vermeiden, muss darauf geachtet werden, dass das Layout bei diesem Vorgang nicht verrutscht und die Glasplatte sauber ist. Die Vakuumpumpe des Belichtungsgerätes sorgt dafür, dass das Layout nicht verrutschen kann. Der Deckel des Belichtungsgerätes kann anschliessend geschlossen werden, um die Platine eine angemessene Zeit zu belichten. In Abhängigkeit von dem verwendeten Belichtungsgerät, sowie der verwendeten Epoxid-Platine, ergibt sich eine optimale Belichtungsdauer von 2 Minuten und 30 Sekunden.

3.3.3 Entwicklung

Nach der Belichtung kann die Platine entwickelt werden um die belichteten Lackteile abzulösen. Dazu wird eine Natriumhydroxid-Lösung, mit einem Mischungsverhältnis von 1:100, verwendet. Die Platine kann nach etwa einer halben Minute aus der Lösung genommen und anschliessend mit Wasser abgespült werden. Das komplette Layout der Platine sollte sichtbar sein. Es dürfen keine Lackreste mehr vorhanden sein, ansonsten muss der Vorgang wiederholt werden. Ferner sollten die Platinen an den Kupferstellen glänzen an denen der Fotopositivlack abgelöst wurde. Diese Stellen werden beim Eintauchen in das Ätzbad sofort matt, was den beginnenden Ätzprozess zeigt. Ist dies nicht der Fall, muss die Entwicklung wiederholt werden. Die fertig entwickelte Platine sollte dann auf unterbrochene Verbindungen kontrolliert werden, diese können zum Beispiel mit einem wasserfesten Stift repariert werden.

3.3.4 Ätzen

Nach dem Entwickeln der Platine folgt das Wegätzen der freigelegten Kupferschicht. Als Ätzmittel wurde Natriumpersulfat eingesetzt. Das Ätzmittel muss vor Beginn, mit Hilfe eines Heizstabes, auf eine Temperatur von mindestens 40 Grad Celsius und höchstens 60 Grad Celsius gebracht werden. Zusätzlich wird in den Ätzbehälter von unten Luft geblasen um den Ätzvorgang zu beschleunigen. Nach einer ungefähren Dauer von 5-10 Minuten ist die Platine fertig geätzt und kann mit Wasser abgespült werden.

3.3.5 Säubern und Versiegeln

Nach dem Ätzen sind auf der Platine noch Rückstände von Fotopositivlack vorhanden. Dieser kann mit Spiritus entfernt werden. Die gesäuberte Platine wird abschliessend mit Lötack besprüht. Die Versiegelung verhindert die Oxidation der Leiterbahnen.

3.3.6 Bohren

Ist die mit Lötack behandelte Platine trocken, kann gebohrt werden. Es wurde ein Hartmetallbohrer verwendet, da normale High-Speed-Steel-Bohrer (HSS-Bohrer) bei dem Einsatz mit Glasfaserplatten schnell brechen können. Der Bohrer hatte weiterhin den Vorteil, dass er an einem Bohrständler montiert war.

3.3.7 Löten

Im letzten Vorgang werden die Bauelemente, mit Hilfe eines Lötkolbens, auf die Platine gelötet. Beim Lötvorgang wird Lötzinn erhitzt. Das Flussmittel, das im Lötendraht enthalten ist, reduziert die Oxidation der metallischen Oberflächen und gewährleistet eine zuverlässige mechanische und elektrische Verbindung zwischen den Metallteilen. Fehler beim Löten können mit Hilfe einer Entlötpumpe korrigiert werden. Diese dient zum Absaugen von flüssigem Lötzinn beim Lösen einer Lötverbindung.

3.4 Hauptplatine

Die Hauptplatine ist die zentrale Platine des Staubsaugerroboters. Auf ihr befinden sich die meisten Bauteile. Zusätzlich gibt es noch weitere Platinen, die die Bauteile enthalten, die an die Karosserie angebracht werden müssen. Der Schaltplan A.7 der zentralen Platine befindet sich im Anhang. Da die Hauptplatine umfangreich ist, wurden zur Erklärung einige Ausschnitte aus der logischen Ansicht der Schaltung extrahiert. Auf das Kapitel 2 aufbauend, wird hier die Elektronik der Bauelemente behandelt. Die Platine wurde bislang noch nicht getestet, d.h. es könnten Fehler vorhanden sein, die am Anfang des zweiten Semesters korrigiert werden. Zur Zeit wird das Layout für das anschliessende Ätzen und Bestücken vorbereitet. Wie die einzelnen Platinen hergestellt werden, ist im vorherigen Abschnitt 3.3 beschrieben.

3.4.1 ATmega32

Der Mikrocontroller besitzt 32 I/O-Pins verteilt auf vier Ports:

Port A PA0 bis PA7

Port B PB0 bis PB7

Port C PC0 bis PC7

Port D PD0 bis PD7

In den nachfolgenden Abschnitten wird erwähnt, welcher Pin des Mikrocontrollers wie belegt ist und mit welchem Bauteil er verbunden wird. Zum besseren Verständnis des weiteren Textes folgt eine Liste mit der Beschreibung der Pinbelegung. Für weitere Details des Bausteins wird hier auf den Abschnitt 3.2 verwiesen.

Reset Setzt den Controller zurück, wenn ein LOW-Pegel für mindestens zwei Zyklen des Systemtaktes anliegt

XTAL1, XTAL2 Anschluss eines externen Taktgebers (hier: Quarz)

AVCC Betriebsspannung für den Analog-Digital-Wandler

AGND Alternative Masse

AREF Referenzspannung für den Analog-Digital-Wandler

VCC Versorgungsspannung

GND Masse

PA0 bis PA7 ADC0 bis ADC7 sind Eingabepins für analoge Messungen

PB0 T0 kann als Taktgeber für die Zähler 0 und 1 verwendet werden,
TCK als externer Taktpin für synchrone USART-Übertragung

PB1 T1 kann als Taktgeber für die Zähler 0 und 1 verwendet werden

PB2 AIN0 ist ein externer Eingang des Analog-Komparators,
INT2 ist eine externe Interrupt-Leitung

PB3 AIN1 folgt analog zu AIN0,
OC0 kann als Ausgang für Zähler0 verwendet werden (Beispiel: PWM-Modus)

PB4 SS gehört zum SPI-Interface, dieser wird benötigt, um den richtigen Slave am Bus zu wählen

PB5 MOSI gehört zum SPI-Interface, dieser Datenausgang (als Master) oder Dateneingang (als Slave) wird bei ISP verwendet

PB6 MISO gehört zum SPI-Interface, dieser Dateneingang (als Master) oder Datenausgang (als Slave) wird bei ISP verwendet

PB7 SCK gehört zum SPI-Interface, dieser Bustakt vom Master wird bei ISP verwendet

PD0 RX ist Eingang der seriellen Schnittstelle (USART)

PD1 TXD ist der Ausgang für die Serielle Schnittstelle und externe Takt für den USART

PD2, PD3 INT0, INT1 sind externe Interrupt-Leitungen

PD4, PD5 OC1A, OC1B können als Ausgänge für den Zähler1 verwendet werden, beide Pins sind an einem Zähler, es sind aber dennoch verschiedene Signale möglich (Beispiel: PWM)

PD6 ICP1 kann zusammen mit Zähler1 verwendet werden um den aktuellen Wert einzufangen

PD7 OC2 kann als Ausgang für Zähler2 verwendet werden

PC0 SDA ist die Datenleitung der I^2C -Schnittstelle

PC1 SCL ist die Clockleitung der I^2C -Schnittstelle

PC2, PC3, PC4, PC5 TCK, TMS, TDO und TDI gehören zum JTAG-Debug Interface, worüber der Mikrocontroller programmiert und debuggt werden kann

PC6, PC7 TOSC1, TOSC2 kann als externer Taktgeber für Zähler2 angeschlossen werden

Zusammenfassend können die 32 I/O-Pins in acht verschiedene Bereiche eingeteilt werden:

1. Analog-Digital-Wandler (PA0, PA1, PA2, PA3, PA4, PA5, PA6, PA7)
2. Analog-Komparator (PB2, PB3)
3. SPI-Schnittstelle (PB4, PB5, PB6, PB7)
4. Timer und PWM (PB0, PB1, PD4, PD5, PD6, PD7)
5. I^2C -Schnittstelle (PC0, PC1)
6. JTAG-Interface (PC2, PC3, PC4, PC5, PC6, PC7)
7. Serielle Schnittstelle (PD0, PD1)
8. Externe Interrupts (PD2, PD3)

3.4.2 Quarz

Der ATmega32 benötigt ein Quarz-Bauelement mit einer Taktfrequenz von 16Mhz. Damit wird festgelegt, wie schnell das Bauteil läuft. Obwohl der ATmega32 einen internen Oszillator besitzt, dient der externe Quarz dazu, dem Mikrocontroller einen genaueren Takt zu geben. Ein Pin vom Quarz ist mit dem XTAL1-Eingang des Mikrocontrollers verbunden und der andere Pin mit dem XTAL2-Eingang. Die Polung des direkt anschließbaren Quarzes ist irrelevant. Beide Pins des Quarzes werden zusätzlich über einen $22pF$ -Kondensator mit der Masse verbunden

3.4.3 Kompassensor und Gumstix

Abbildung 3.15 zeigt, dass Kompassensor und Gumstix über eine Steckverbindung an die Hauptplatine anzuschließen sind. Hierfür werden die Pins PC1 (SDA) und PC0 (SCL) benutzt. Beide Leitungen sind über PullUp-Widerstände mit einem Lötpad verbunden. Das Pad stellt einen Platzhalter zum löten dar. Beide Widerstände besitzen $47k\Omega$. Das Pad bekommt über gelötetes Kabel eine Spannung von 3,3V vom Gumstix. Die Vier-Pin-Steckverbindung führt zum Gumstix und versorgt diesem mit 5V von der Spannungsquelle.

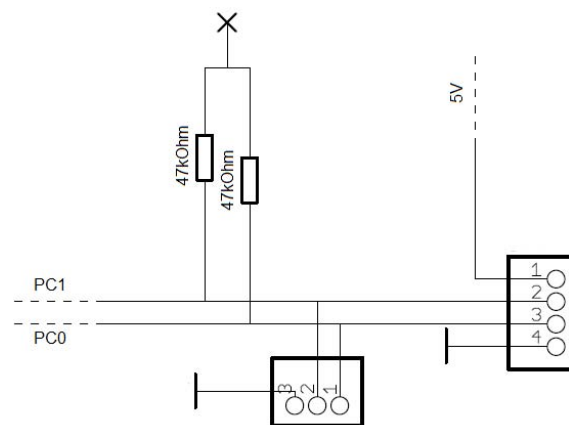


Abbildung 3.15: Realisierung der Hauptplatine - Kompassensor und Gumstix

3.4.4 Reset

Es ist möglich den Roboter durch einen Taster neu zu starten. Abbildung 3.16 zeigt, wie dieser an den invertierenden RESET-Pin des Mikrocontrollers angeschlossen ist. Ist der Taster gedrückt, wird GND durchgeschaltet und der Vorgang ausgelöst. Über einen $10k\Omega$ PullUp-Widerstand bekommt der Taster eine Spannung von 5V. Dies ist nötig, damit er bei unbeschaltetem Zustand einen HIGH-Pegel hat und sich nicht zurücksetzt. Der Kondensator hat eine Kapazität von $100nF$.

3.4.5 Radencoder

Abbildung 3.17 zeigt den Schaltplan der beiden Radencoder. An die Pads werden die Lichtsensoren angeschlossen, d.h. jeweils ein Pad rechts und links pro Sensor. Da die Bauelemente CNY70 sich sehr nah an den Drehscheiben der Räder befinden müssen, wurde für jeden dieser Sensoren eine gesonderte Platine erstellt. Der Schaltplan (A.8) hierzu befindet sich im Anhang. Die Vorwiderstände der IR-LED des CNY70 besitzen 100Ω . Die Widerstände für den IR-Fototransistor haben 680Ω . Die beiden Pullup-Widerstände vor den beiden invertierenden Schmitttriggern betragen $47k\Omega$. Sie sind an die 5V Spannungsquelle

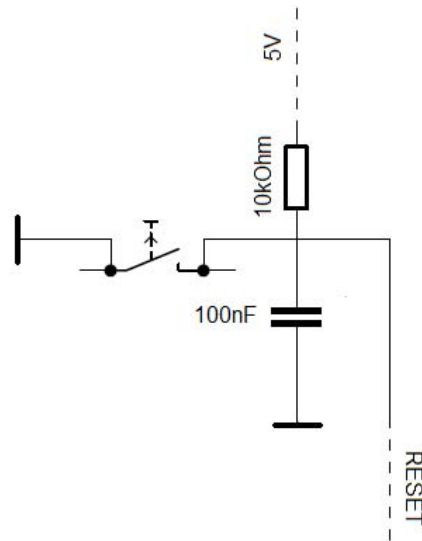


Abbildung 3.16: Realisierung der Hauptplatine - Reset

angeschlossen. Die beiden Schmitttrigger geben ihr Ausgangssignal an den Mikrocontroller weiter. Der obere ist mit dem Pin PD2 (INT0) verbunden, der untere mit PD3 (INT1).

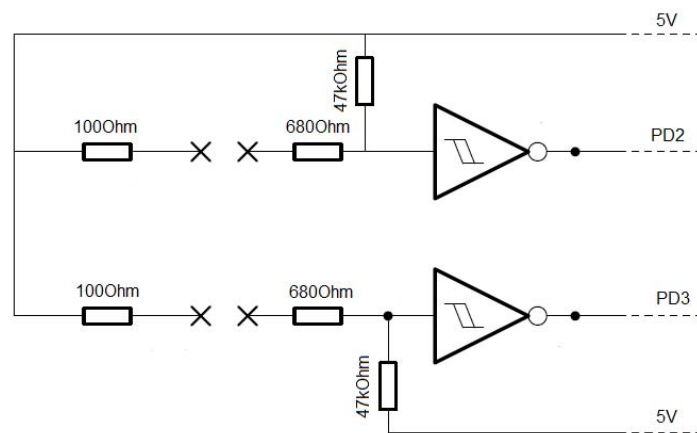


Abbildung 3.17: Realisierung der Hauptplatine - Radencoder

3.4.6 Ultraschallsensoren

Wie in Abschnitt 2.1.2 des zweiten Kapitels bereits erwähnt, werden die Ultraschallsensoren im Modus 2 betrieben. Das bedeutet, dass diese eine gemeinsame Trigger- und Echoleitung benutzen. Hierfür werden die zwei entsprechenden Pins auf GND gesetzt, die die Hauptplatine zur Verfügung stellt. Ebenfalls wird eine 5V Spannungsquelle für einen Pin

bereitgestellt. Die vier Ultraschallsensoren, bzw. dessen Trigger-/Echo-Leitungen sind an die Pins PA3 (ADC3), PA4 (ADC4), PA5 (ADC5) und PA6 (ADC6) angeschlossen.

3.4.7 Leuchtdioden

RAPTOR besitzt mehrere verschiedene LEDs, die unterschiedlichen Zwecken dienen:

- Betriebszustand
- Akkumulatorzustand
- Bodenbeleuchtung

Für den Betriebszustand und den Akkumulatorzustand existiert eine gemeinsame Zusatzplatine, siehe Abbildung (A.10) im Anhang. Die Vorwiderstände beider 10mm großen LEDs betragen 160Ω . Die Widerstände befinden sich auf der Hauptplatine. Durch die Pads werden auch hier Zusatzplatine und Hauptplatine an den entsprechenden Stellen verdrahtet und verlötet. Wird der Roboter in Betrieb genommen, signalisiert er dies mit einer blauen LED an dem Gehäuse.

Der Nutzer wird über eine rote LED informiert, wenn die Akkuleistung sinkt. Diese Information wird benötigt, da der Prototyp keine Aufladestation hat, die er selbstständig anfahren könnte.

Damit die Kamera genügend Licht zum Fotografieren hat, besitzt der Staubsaugerroboter auf seiner untersten Ebene sechs Leuchtdioden. Da dies aufgrund der nötigen Bodennähe nicht auf der Hauptplatine implementiert werden kann, existiert hierfür eine Zusatzplatine. Der Schaltplan befindet sich im Anhang A.9. Der Pin PB3 (AIN1/DC0) des ATmega32 benutzt einen Vorwiderstand von 100Ω zum N-MOSFET-Transistor. Wenn das Licht der Bodenbeleuchtung für die Kamera benötigt wird, wird der Transistor vom Mikrocontroller in Durchlassrichtung gesetzt und die Zusatzplatine die mittels eines Kabels mit der Hauptplatine verbunden ist, erhält die benötigte Spannung. Die sechs LEDs dort haben jeweils einen Vorwiderstand von 160Ω . Die LEDs haben jeweils einen Durchmesser von 5mm.

3.4.8 Motoren

Der RAPTOR besitzt vier Getriebemotoren: zwei für die Räder und zwei für die vorderen Bürsten. Abschnitt 3.1.3 beschreibt, wie die Testplatine für die Motorsteuerung entstand. Die gewonnenen Erkenntnisse und Erfahrungen mit dem Baustein L293D konnten bei der Benutzung mit der Hauptplatine größtenteils übernommen werden. Im Folgenden wird näher auf die Realisierung der Motoren auf der Hauptplatine eingegangen.

Bürsten

Die vorderen beiden Bürsten haben eine festgelegte Motordrehrichtung: die rechte Bürste dreht gegen den Uhrzeigersinn, die linke Bürste im Uhrzeigersinn. Damit drehen beide

Bürsten nach innen um den Schmutz in die Richtung der Saugöffnung zu befördern. Abbildung 3.18 zeigt, wie zum Festlegen der Drehrichtung zwei Inverter verwendet worden sind, damit es nicht mehr nötig ist, dieses per Software zu steuern. Zusätzlicher Vorteil ist, dass dafür keine Pins am ATmega32 belegt werden. Der Pin 1A ist mit dem Eingang eines der beiden Inverter verbunden und invertiert sein Signal nach Pin 2A des Bausteins L293D. Der Eingang des zweiten Inverters ist mit dem Pin 3A verbunden. Der Pin 4A erhält dieses invertierte Signal. Die beiden Eingänge der Inverter wurden jeweils mittels einer Leitung zusammengeführt und mit Pin PD6 (ICP) des Mikrocontrollers verbunden. Die Pads der Pins 1Y und 2Y werden über Kabel an den ersten Motor gelötet, die Pins 3Y und 4Y an den zweiten Motor. Die Anschlüsse GND1, GND2, GND3 und GND4 werden geerdet. Da sich die Motoren der Bürsten nicht so schnell drehen müssen, wie die Motoren der Räder, erhalten VCC1 und VCC2 jeweils eine Spannung von 5V. Die Pins 1-2EN und 3-4EN vom Motortreiber-IC sind ebenfalls beide mit der 5V-Spannungsquelle verbunden. Auch hier ist der Vorteil, dass keine zusätzliche Steuerung per Software erfolgen muss und auch keine zusätzlichen Pins des Atmel-Bausteins belegt sind. Für den autonomen Staubsaugerroboter bedeutet dies allerdings, dass sobald er in Betrieb genommen wird, am Motortreiber-IC Spannung anliegt und die Bürsten sich solange drehen, bis sich der Betriebszustand auf ausgeschaltet ändert.

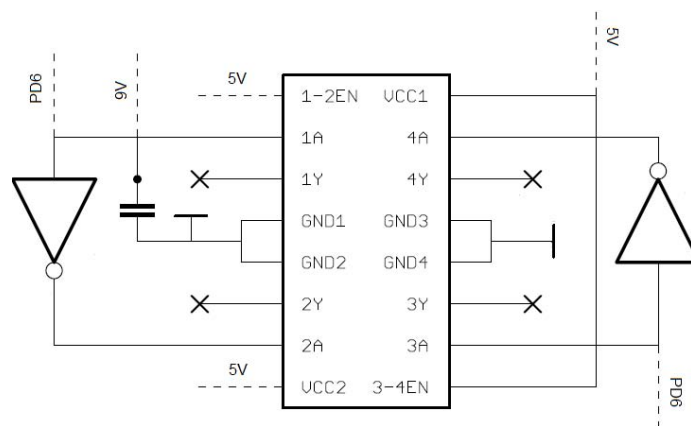


Abbildung 3.18: Realisierung der Hauptplatine - Bürstenmotor

Räder

Für die Motorsteuerung des Rades ist wichtig zu beachten, dass der Roboter sowohl rückwärts, als auch vorwärts fahren kann. Anders als bei der Ansteuerung des Motortreiber-ICs für die Bürsten, ist hier die Drehrichtung des Motors nicht festgelegt. Wie Abbildung 3.19 zeigt, sind die Pins GND1, GND2, GND3 und GND4 auf den logischen Wert Null

gesetzt. Die Pins 1Y und 2Y werden an den ersten Motor der Räder befestigt, die Pins 3Y und 4Y an den zweiten Motor der Räder. Der L293D erhält am Pin VCC1 5V Spannung, für die logische Schaltung, und an VCC2 9V Spannung für die Räder. An VCC2 könnten auch bis zu 12V angeschlossen werden, jedoch sollte der Motor auf Dauer nicht mit der Maximalleistung betrieben werden. Um den Motor über die Software ansprechen zu können, besitzt er vier Anschlusspins am ATmega32. Pin 1A kommuniziert mit PC7 (TOSC2) des Mikrocontrollers, Pin 2A mit PC6 (TOSC1), Pin 4A mit PC5 (TDI) und Pin 3A mit PC4 (TDO). Für die Pulsweitenmodulation kommuniziert 1-2EN des Motortreiber-IC's mit PD5 (OC1A) des Mikrocontrollers. Der Pin 3-4EN ist mit PD4 (OC1B) verbunden.

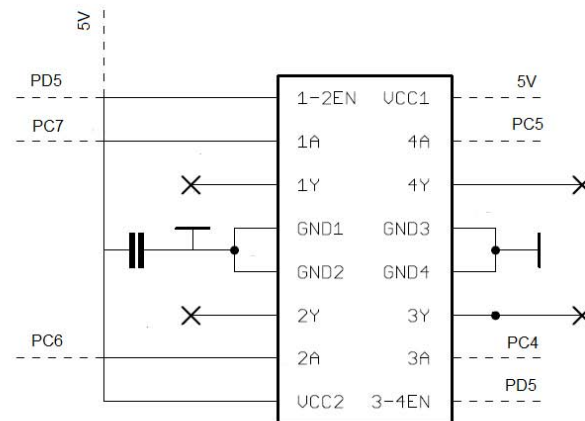


Abbildung 3.19: Realisierung der Hauptplatine - Radmotor

3.4.9 Akkumulator

Der Akku Kung Long der gleichnamigen Firma [15] ist ein wartungsfreier Blei-Gel-Akku mit einer Spannung von 12 V und 4,5 Ah. Abbildung 3.20 zeigt links die Steckverbindung, woran der Akku mit der Hauptplatine verbunden wird. Die beiden Pads rechts daneben sind für den An- bzw. Ausschalter des Roboters. Dieser ist ein Schiebeschalter, der an dem Gehäuse befestigt wird. Die Schottky-Diode verhindert einen Rückfluß des Stroms. Des Weiteren sind zwei Spannungswandler zu sehen. Diese sind nötig, um die 12V Spannung zuerst in 9V Spannung zu übersetzen und anschließend in 5V Spannung. Die 9V-Spannung brauchen nur die Motoren der Räder, die übrigen Bauelemente der Hauptplatine benutzen eine Spannung von 5V. Bei der Übersetzung wird Hitze erzeugt die durch einen Kühlkörper abgeführt werden muss. Die Eingänge der beiden Spannungswandler haben Kondensatoren mit jeweils $1000\mu F$. Die Kondensatoren an den Ausgängen besitzen $100nF$. Abschließend

sei noch zu erwähnen, dass sich unterhalb der Pads ein Spannungsteiler befindet mit einer Reihenschaltung zweier Widerstände. Der Widerstand nach GND benötigt $1k\Omega$, der andere Widerstand $2k\Omega$. Der Spannungsteiler ist mit dem Pin PA7 (ADC7) des ATmega32 verbunden. Ist der Akku fast verbraucht, sollte der RAPTOR ausgeschaltet werden um Schäden am Bauteil zu vermeiden.

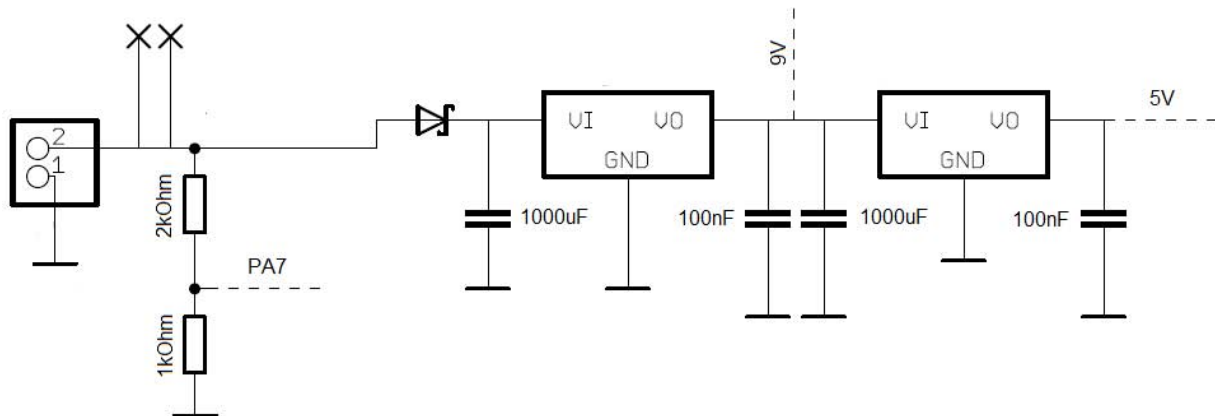


Abbildung 3.20: Realisierung der Hauptplatine - Akkumulator

3.4.10 Kontaktsensoren

Wie in Abschnitt 2.1.3 schon erwähnt, ist der vordere Bereich des Staubsaugerroboters mit fünf Kontaktsensoren (sogenannte Bumper) ausgestattet. Falls der autonome Roboter auf ein Hindernis trifft, wird dem Mikrocontroller dies mitgeteilt, so dass dieser entsprechend reagieren kann. Bei dieser Implementierung kann nicht erkannt werden, welcher Bumper ausgelöst wurde. Im idealen Fall sollte der RAPTOR es nicht zu einer Kollision kommen lassen, da die Ultraschallsensoren Hindernisse vorher erkennen sollten. Die Bumper sind demzufolge eine Absicherung gegen eine Fehlfunktion der Ultraschallsensoren. In Abbildung 3.21 ist zu sehen, wie jeweils zwei Pins der fünf Kontaktsensoren über ein Kabel an die Pads der Hauptplatine gelötet werden. Das jeweils linke Pad (PAD24, PAD25, PAD26) auf dem Schaltentwurf ist mit GND verbunden. Die rechten Pads (PAD27, PAD28, PAD39) sind mit 5V Spannung versehen und haben einen Vorwiderstand der jeweils $1k\Omega$ beträgt. Sollte ein Kontaktsensor ausgelöst werden, setzt sich die logische Null der linken Seite gegenüber der logischen Eins der rechten Seite durch. Sobald eine Null an dem UND-Bauteil anliegt, wechselt das Signal zu Null. Das jeweilige Signal wird mit dem Pin PB2 (AIN0/INT2) des ATmega32 verbunden.

3.4.11 Abgrundsensoren

Für die Abgrunddetektion werden ebenfalls Lichtsensoren vom Typ CNY70 verwendet. Wie bei den Radsensoren der Hauptplatine, müssen die Sensoren sehr nah am zu messenden

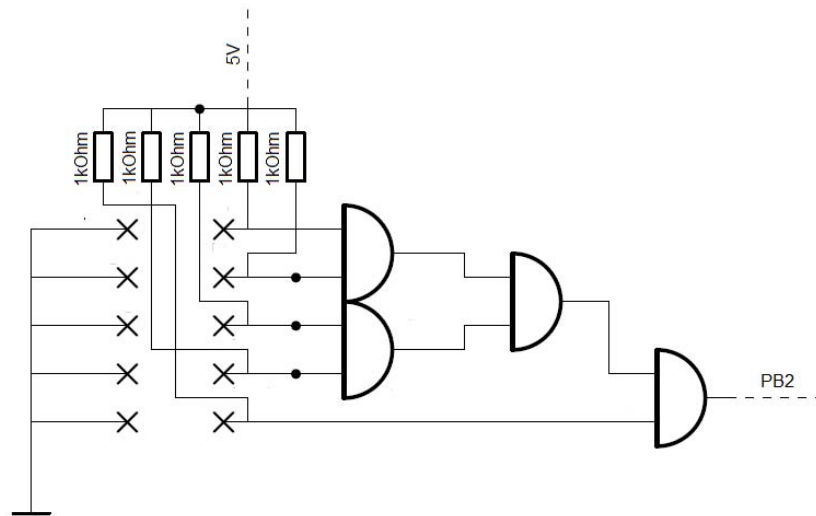


Abbildung 3.21: Realisierung der Hauptplatine - Kontaktsensoren

Gegenstand sein. In diesem Fall müssen sie sehr nah am Boden angebracht werden. Die drei Lichtsensoren befinden sich auf Zusatzplatinen. Mit Hilfe der Pads werden sie an die Hauptplatine über Kabel gelötet. Die Vorwiderstände betragen auch hier 100Ω und die PullUp-Widerstände $47k\Omega$. Alle drei sind mit 5V Spannung versehen. Der im Schaltplan oberste CNY70 ist somit mit dem Pin PA0 (ADC0) des ATmega32 verbunden, der mittlere mit PA1 (ADC1) und der unterste mit PA2 (ADC2).

3.4.12 Maussensor

Für die korrekte Ermittlung der abgefahrenen Strecke muss der optische Maussensor sehr nah am Boden angebracht werden. Aus diesem Grund wurde der Maussensor auf einer eigenen Platine untergebracht. Als Grundlage für die Schaltung diente der Schaltplan vom c't-Bot [7], welcher entsprechend modifiziert wurde. Im Anhang befindet sich der neue Schaltplan A.6 des Maussensors. Die Verbindung zum Mikrocontroller erfolgt über einen vierpoligen Stecker, welcher Anschlüsse für 5V, Ground, SDIO (Datenleitung) und SCK (Taktleitung) bereitstellt. An die beiden Pins OSC-IN und OSC-OUT des Sensorchips wird der Quarz angeschlossen, welcher den Takt für die Bildaufnahmen vorgibt. Die 5V-Leitung wird an den VDD-Pin des Chips geführt, außerdem speist sie über einen Vorwiderstand von 100Ω die Leuchtdiode. Der zweite Pin der Diode wird an den BC557-Transistor angeschlossen. Die Schaltung des Transistors wird über den LED-CNTL-Pin gesteuert. Schaltet der Transistor, so liegt am zweiten Pin der Diode Ground an. Des Weiteren wird die Ground-Leitung über einen $1\mu F$ -Kondensator an den REFA-Pin angeschlossen, und direkt an den GND-Pin des Sensorchips. Ein weiterer $100nF$ -Kondensator befindet sich zwischen den Ground und 5V-Leitungen. Die SDIO- und SCK-Leitungen werden direkt vom Stecker an die entsprechenden Pins des Chips geführt.

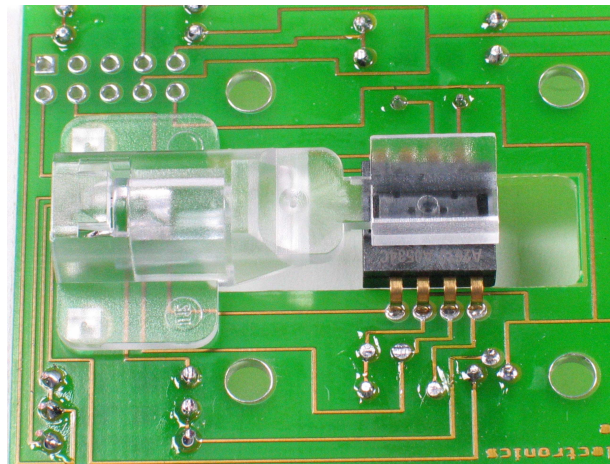


Abbildung 3.22: Der optische Maussensor [7]

3.5 Karosserie

Nach einigen Überlegungen fiel die Entscheidung auf einen Roboter, der aus zwei Ebenen besteht. So ist es möglich, alle Komponenten problemlos unterbringen zu können. Die Ebenen haben eine runde Form. Dadurch wird das Drehen auf der Stelle ohne Anecken ermöglicht. Auf den Ebenen werden die zuvor vorgestellten Bauteile befestigt. Die untere Ebene soll unter anderem die Motoren, den Akku und die Befestigung der Räder beherbergen. Auf der oberen Ebene wird beispielsweise der Gumstix sowie einige Sensoren angebracht. Für den Aufbau der zwei Ebenen haben wir uns für zwei runde Aluminiumplatten mit einem Durchmesser von 30cm entschieden und diese vom Institut für Spanende Fertigung an der Technischen Universität Dortmund herstellen lassen. Die beiden Platten werden durch vier Metallstangen mit einem Durchmesser von 0,5 cm verbunden. Damit steht das Grundgerüst des Roboters. Die Radkonfiguration sieht zwei normale Räder plus ein Kugelrad vor, die an die untere Ebene montiert werden. Die zwei normalen Räder sind die eines Tretrollers. Der Durchmesser beträgt jeweils 10 cm und die Breite 2,5 cm. Die Räder werden auf entgegengesetzten Seiten der Mittelachse der Kreisscheibe angebracht. Dadurch kann sich der Roboter auch auf der Stelle drehen. Als Kugelrad fungiert ein Deoball, der nur eine Funktion als Stützrad hat und im Gegensatz zu den zwei normalen Rädern nicht angetrieben wird. Damit die Ecken eines Raumes auch gut gesäubert werden können, wurden zwei kleine Bürsten mit einem Durchmesser von 5cm gekauft. Diese werden vorne an der unteren Platte befestigt. Als Motoren für die Räder und Bürsten kommen jeweils die gleichen vier Motoren mit 12V zum Einsatz. Die folgenden Abschnitte erklären ausführlich die bisherigen Fertigungsschritte und den Einbau aller bis zum Zwischenbericht angebrachten Teile. Zuletzt werden noch die Konstruktionspläne mit den genauen Angaben in mm angegeben. Die Pläne enthalten alle bisher eingebauten Komponenten, sowie einige der geplanten Komponenten. Die Konstruktionspläne sind in Anhang A zu finden.

3.5.1 Die untere Ebene

In diesem Abschnitt wird erklärt, wie die Räder, Motoren, Bürsten, das Kugelrad, Akku und die Stangen befestigt wurden und wie die Platte dafür bearbeitet werden musste.

Bearbeitung der unteren Platte

Bevor mit der maschinellen Bearbeitung der Platte begonnen werden konnte, wurden mit Hilfe eines Zirkels zwei rechtwinklige Diagonalen durch den Mittelpunkt gezeichnet. Anhand dieser Diagonalen wurden dann die genauen Positionen für die folgenden Bearbeitungsschritte festgelegt. Mit Hilfe der Fräsmaschine, am Lehrstuhl 12 der Technischen

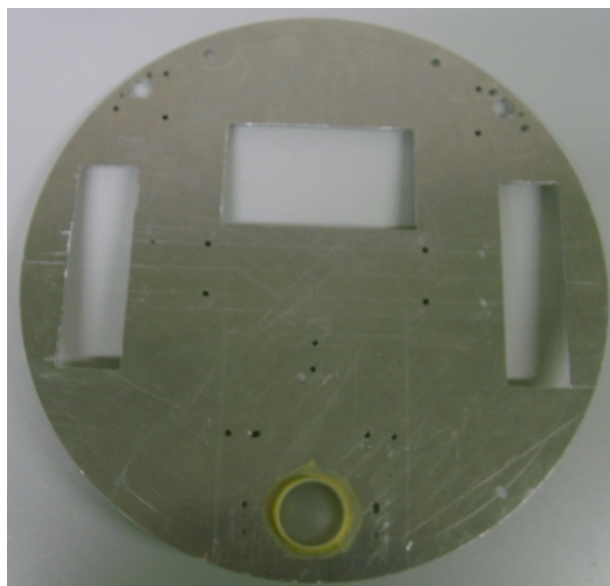


Abbildung 3.23: Bearbeitung der unteren Aluminiumplatte

Universität Dortmund, wurden als erstes die Löcher für die Räder ausgefräst. Die Löcher sind jeweils 10,7 cm lang und 3,2 cm breit. Der Abstand eines Lochs vom Rand der Platte beträgt an der Diagonalen 2,5 cm. Danach wurde, weiterhin mit dem Fräser, das Loch für die Kamera ausgeschnitten. Das Loch für die Kamera ist 5,5 cm lang und 10cm breit. Für die Befestigung der Bürsten an die Motoren musste folgendes gemacht werden. Es wurden für jeden Motor jeweils vier Löcher gebohrt, nämlich ein Loch für die Achse des Motors, der später an den Bürsten befestigt wird und drei Löcher für die Befestigung der Motoren an die Platte. Da die genauen Positionen der drei Löcher zur Motor - Aluplattenbefestigung schwierig auszumessen sind, wurde dafür eine Schablone erstellt. Für das Kugelrad (den Deoball) wurde im hinteren Teil der Platte, mit einem Spezialwerkzeug für die Fräsmaschine, ein Kreis herausgeschnitten, in den der Deoball gesteckt wird. Zur besseren Befestigung wurde der Deoball zusätzlich an der Aluplatte festgeklebt. Das Ergebnis der bisherigen Schritte ist in Abbildung 3.23 zu sehen. Nachdem diese Löcher in die untere Platte gefräst wurden, wurde mit der Bearbeitung der Räder begonnen.

Befestigung Räder an den Motoren

Um die Räder an den Motoren und der Platte zu befestigen, wurde ein zylinderförmiger Metallblock mit einer Handsäge, und danach zur Glättung mit der Drehmaschine, auf die Radbreite gebracht. Der Durchmesser hatte schon die richtige Größe. Die klein geschnittenen Zylinderstücke passten damit genau in die Löcher der Räder. Auf einer Seite der Zylinder wurde ein Loch für die Motorachse gebohrt. Senkrecht zu diesem Loch wurde noch ein kleineres Loch von 3mm Durchmesser gebohrt, wodurch der Motor an den Zylinder geschraubt werden konnte (Abbildung 3.24).

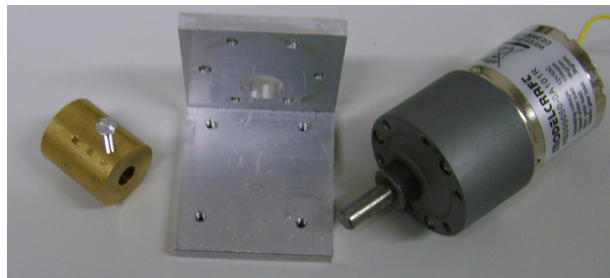


Abbildung 3.24: Bearbeitung der unteren Aluminiumplatte

An den Rädern wurden kleine Stücke mit der Fräsmaschine ausgeschnitten. Die Schraube zur Motor - Zylinderbefestigung dient damit gleichzeitig als Zylinder - Radbefestigung. Damit waren die Räder mit den Motoren verbunden (Abbildung 3.25). Als nächstes mussten die Motoren dann mit der unteren Ebene verbunden werden.

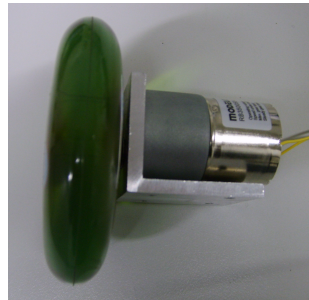


Abbildung 3.25: Bauteile zur Motormontage

Befestigung der Räder an die untere Platte

Am LS-12 der Technischen Universität Dortmund, waren L-förmige Metallstücke vorhanden, die durch Fräsarbeiten auf die im Anhang angegebene Größe gebracht wurden. In eine Seite des rechtwinkligen Metallstücks wurden die Löcher zur Motorbefestigung gebohrt - analog zu den Löchern in der unteren Ebene. Dann wurden sowohl auf der anderen Seite

des Winkels als auch auf der Platte selbst vier kleine Löcher gebohrt. Mittels Schrauben konnte dann der Winkel an der Platte befestigt werden.

Als nächstes wurden erst einmal die Motoren an die L-Formigen Metallstücke geschraubt und dann die Motoren mit den Metallstücken an den vorher bearbeiteten Zylinder geschraubt. Danach wurden die Zylinder in die Löcher der Räder gesteckt. Zuletzt wurden dann die L-Formigen Metallstücke mit den Motoren und Rädern an die Platte geschraubt (Abbildung 3.26).

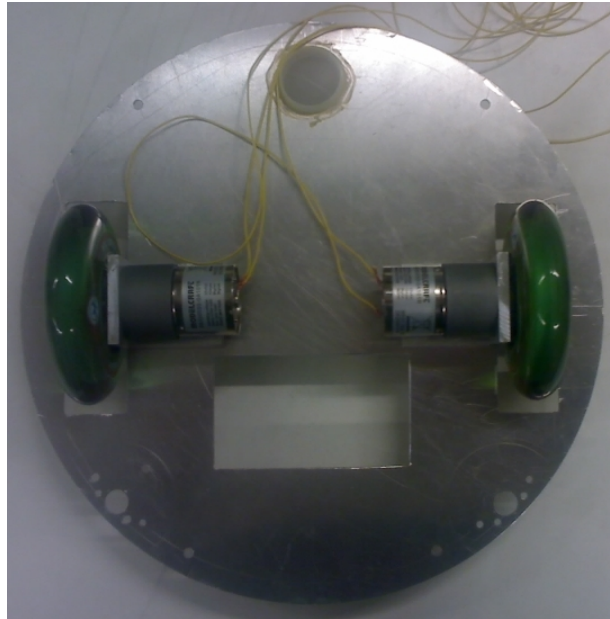


Abbildung 3.26: Befestigung der Räder an der unteren Platte

Befestigung des Kugelrads an die untere Platte

Damit der Roboter später stabil auf dem Boden steht, wurde im hinteren Abschnitt der Platte ein Deoball als Kugelrad befestigt. Bei der Bearbeitung der unteren Platte wurde schon das Loch ausgeschnitten, in dem der Ball angebracht wird. Nach einigen Überlegungen wurde die Entscheidung getroffen, den Deoball an die Platte zu kleben.

Befestigung des Akkus

Zur besseren Gewichtsverteilung wurde der Akku im hinteren Bereich der unteren Ebene angebracht. Damit die zweite Ebene möglichst tief angebracht werden kann, wurde der Akku auf der unteren Ebene liegend montiert. Da der Deoball auch im hinteren Teil befestigt ist, bestand die Notwendigkeit, den Akku etwas höher zu legen, was aber insgesamt immer noch tiefer war, als den Akku stehend zu montieren.

Ein am Lehrstuhl vorhandenes rechteckiges Metallstück wurde durch Fräsen in eine L-Form gebracht, ähnlich wie bei der Motorfestigung der Räder. Danach wurden Stücke mit

der Handsäge klein gesägt und Löcher zur Befestigung in die Metallstücke und die Aluplatte gebohrt (Abbildung 3.27).

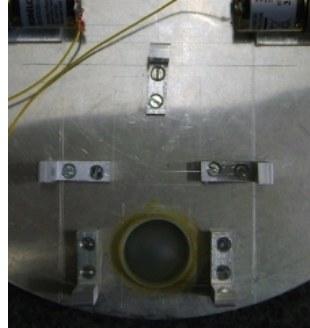


Abbildung 3.27: Befestigung des Akkus an der unteren Platte

Die Metallstücke wurden dann an der Platte befestigt und der Akku eingesetzt. Durch die Platzierung der Metallstücke sitzt der Akku fest auf der unteren Ebene. Mit der oberen Aluminiumplatte wird eine Verschiebung des Akkus auch nach oben verhindert, womit er letztendlich fest sitzt und sich nicht mehr verschieben kann.

Befestigung der Bürsten an der unteren Platte



Abbildung 3.28: Bürsten

An den Bürsten (Abbildung 3.28) wurde jeweils ein Loch für die Befestigung der Motoren gebohrt. Dann wurden die Motoren für die Bürsten an die Platte geschraubt und die Bürsten an den Motoren befestigt (Abbildung 3.29).

Befestigung der Stange an der unteren Platte

Es wurden 5mm dicke Stangen gekauft und diese in 15cm lange Stücke gesägt. Durch zuvor gebohrte Löcher in den Aluminiumplatten und mit Hilfe von Muttern wurden die Stangen mit den Platten und damit letztendlich beide Platten miteinander verbunden (Abbildung 3.30).

3.5.2 Die obere Ebene

In diesem Abschnitt wird die Bearbeitung der oberen Ebene beschrieben. Dazu gehört die Befestigung der Ultraschallsensoren und das Anbringen der oberen Platte an den Stangen.



Abbildung 3.29: Befestigung der Bürsten an der unteren Platte



Abbildung 3.30: Befestigung der Stange an der unteren Platte

Ultraschallsensoren

Wiederrum fiel die Entscheidung auf L-Formige Metallstücke, welche durch Sägen und Fräsen in die richtige Form gebracht wurden. Nach dem Bohren von weiteren Löchern war die Befestigung der Sensoren an den Winkeln und der Winkel an der oberen Ebene möglich (Abbildung 3.31).



Abbildung 3.31: Ultraschallsensor an dem Metallstück

Befestigung der oberen Platte an der Stange

Nachdem die Ultraschallsensoren an der obere Platte befestigt wurden, konnte die Platte an den Stangen befestigt werden (Abbildung 3.32). Dies ist gleichzeitig der zum Zeitpunkt des Zwischenberichts aktuelle Stand der Fertigungsarbeiten.

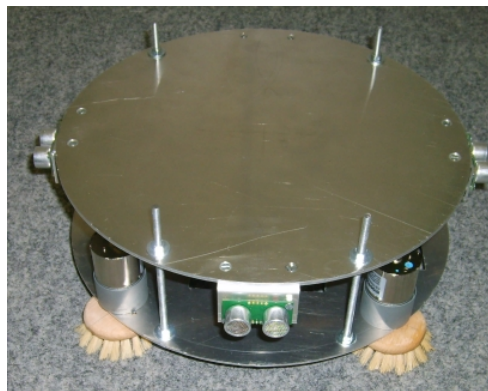


Abbildung 3.32: RAPTOR

4 Software

Nachdem das zweite Kapitel Einblick in die zu programmierende Software gab, spezialisiert sich dieses Kapitel auf die Realisierung der Software für den RAPTOR. Beginnend mit der Software für den Gumstix, wird danach die Programmierung der Gitterkarte vorgestellt. Anschließend wird der Versuch der Benutzung des Voronoi-Diagramms und der daraus resultierenden Probleme geschildert. Des Weiteren wird das Buildroot System und der Umgang mit diesem erklärt. Ferner wird die Simulation mit Player-Stage gezeigt. Anschließend wird der Nutzen von I^2C erläutert, gefolgt von der Beschreibung des ersten API-Entwurfs. Abschließend werden noch die Anfänge der Softwareimplementierung für den ATMega32 beschrieben.

4.1 Gumstix

Der Gumstix besitzt im Auslieferungszustand eine Menge Funktionen, jedoch nicht alle, die für die Nutzung des Raptors wichtig sind. Zum Einen müssen zusätzliche Komponenten, wie das I^2C eingebunden werden, zum Anderen die Funktionen der Erweiterungsboards, wie die Nutzung von LAN, oder des Micro-SD Slots, aktiviert werden. Im folgenden Abschnitt werden diese Änderungen am Gumstix Schritt für Schritt erklärt.

4.1.1 Aufbau des Flash-Speichers

Neben dem 128MB großen Ram besitzt der Gumstix einen 32MB umfassenden Flash-Speicher, welcher in mehrere Sektionen aufteilt ist. Diese werden an dieser Stelle nur kurz aufgelistet, aber nicht näher auf die genauen Spezifikationen erläutert.

- 1:0** In dieser Sektion werden die relevanten Daten und Programme des Bootloaders U-Boot gespeichert. Sollten diese Daten beim Flashen beschädigt werden, wäre es nicht mehr ohne Weiteres möglich den Gumstix zu reparieren.
- 1:1** Diese Sektion ist reserviert für dynamische Daten, wie zum Beispiel Laufzeitvariablen. U-Boot besitzt einen Softwareschutz um nicht zufällig die ersten beiden Sektionen zu überschreiben.
- 1:2** Hier lässt sich das JFFS2 Dateisystem wiederfinden, welches die Root Partition für das Betriebssystem ist. Im Dateisystem ist auch der Linux Kernel enthalten, sowie die Betriebsumgebung.

4.1.2 U-Boot

Bei U-Boot handelt es sich um den Bootloader des Gumstix. Mit diesem gibt es die Möglichkeit ein falsch geflashtes Dateisystem oder einen falsch geflashten Kernel wieder zu reparieren, indem das Dateisystem oder der Kernel noch mal über den Bootloader geflasht wird. Um in die Konsole des Bootloaders zu gelangen, muss sofort nach dem Einschalten des Gumstix eine beliebige Taste gedrückt werden. Sonst wird nach nur 3 Sekunden der Kernel normal gebootet.

4.1.3 Inbetriebnahme

Bevor der Gumstix in Betrieb genommen wird, muss sichergestellt werden, dass eine Stromquelle vorhanden ist, die bei einer Spannung ab 3.3V konstant 600mA Gleichstrom liefert. Ab 4V wird jedoch vermehrt Hitze produziert, was zu Komplikationen führen kann, wenn der Gumstix nicht hinreichend gekühlt wird. Durch die zusätzlichen Geräte, wie SD-Cardreader und Ethernet, hat sich bei geringerer Stromstärke der Gumstix immer wieder nach dem Bootvorgang abgeschaltet.

4.1.4 Konfiguration des Betriebssystems

Als Erstes muss der vom console-vx Zusatzboard bereitgestellte Ethernet-Controller und der micro-SD-Card Slot in Betrieb genommen werden. Diese werden in der Datei `/etc/modules` aktiviert. Hierbei kann auch sofort die Host-Funktion der USB Schnittstelle eingebunden werden. Dazu müssen die jeweils vorhandenen Zeilen durch die folgenden Zeilen ersetzt werden:

```
# MMC support
mmc_block
pxamci

# Ethernet support
smc911x

# USB-Host support
ohci-hcd
```

Listing 4.1: `/etc/modules`

Nach dem nächsten Bootvorgang sollte nun eine micro-SD Karte in das Verzeichnis `/mnt/mmc` gemountet werden. Für die Ethernetnutzung muss in der Datei `/etc/modprobe.conf` noch folgende Zeilen

```
alias eth0 smc91x
alias eth1 smc91x
```

Listing 4.2: `/etc/modprobe.conf`

durch

```
alias eth0 smc911x
alias eth1 smc911x
```

Listing 4.3: /etc/modprobe.conf

ersetzt werden. Als Nächstes wird nun der Ethernettreiber eingebunden. Durch den Kompilervorgang von Buildroot entstehen die in Listing 4.4 aufgeführten Dateien. Diese Dateien müssen nun mit Hilfe einer micro-SD Karte oder einer seriellen Verbindung auf den Gumstix kopiert werden. Die ersten beiden Module müssen in das Verzeichnis `/lib/modules/2.6.21gum/kernel/drivers/net/` und die dritte Datei in das Verzeichnis `/lib/modules/2.6.21gum/kernel/drivers/usb/host/`. Optional kann nun die Bluetooth Funktion, sofern sie nicht genutzt wird, abgeschaltet werden, indem im Verzeichnis `/etc/init.d` die Datei `S30bluetooth` gelöscht oder umbenannt wird. Nach einem Neustart des Gumstix kann nun über den Befehl `ifconfig` festgestellt werden, ob der Ethernet-Controller korrekt eingebunden ist.

```
../gumstix-buildroot/build_arm_nofpu/linux-2.6.21gum/
drivers/net/gumstix-smc911x.ko
../gumstix-buildroot/build_arm_nofpu/linux-2.6.21gum/
drivers/net/smc911x.ko
../gumstix-buildroot/build_arm_nofpu/linux-2.6.21gum/
drivers/usb/host/ohci-hcd.ko
```

Listing 4.4: Ethernet-Treiber-Dateien

4.1.5 Flashen des Kernels/Dateisystems

Um die Software auf dem Gumstix auf dem neuesten Stand zu halten, werden immer mal wieder Änderungen am Kernel und auch am Dateisystem vonnöten sein. Wir haben daher ein Verfahren entwickelt, welches eine sichere Variante für diesen Vorgang darstellt. Im Folgenden wird deshalb das Flashen des Kernel über das Betriebssystem und den Bootloader beschrieben.

Über das Betriebssystem

Natürlich gibt es auch die Möglichkeit einen kompletten, neuen Kernel oder ein anderes Dateisystem auf den Gumstix zu überspielen. Hierfür werden die Dateien benötigt, welche in Listing 4.5 angegeben sind. Diese Dateien werden wieder auf den Gumstix kopiert. Die Befehle `df` oder `mount` sollten anzeigen, in welchem Bereich im Speicher das Root-Image liegt. Das Ergebnis sollte dem Beispiel aus Listing 4.6 ähneln. Sollte aber etwas anderes als `mtdblock1` ausgegeben werden, muss beim Flashen auch eine andere Zieladresse angegeben werden. Zunächst einmal muss jedoch das Dateisystem neu eingebunden werden - allerdings aus Sicherheitsgründen im Nur-Lese-Modus (`ro`). Das eigentliche Flashen geschieht dann

über die Befehls-Reihe, welche in Listing 4.7 aufgeführt ist. Der Watchdog sorgt dafür, dass der Gumstix nach dem Flashen automatisch neu gestartet wird. Am Terminalprogramm sollte der Reboot verfolgt werden können. Sollte alles funktionieren, startet der Gumstix nach der U-Boot-Wartezeit sofort das neue Image.

```
rootfs.arm\_nofpu.jffs2 (das Root-Filesystem)
uImage (der Kernel)
```

Listing 4.5: Kernelimages

```
/dev/mtdblock1 on / type jffs2 (rw,noatime)
```

Listing 4.6: in der Konsole

```
# modprobe sa1100\_wdt margin=1
# mount -o remount,ro /
# echo && flashcp -v /tmp/rootfs.arm_nofpu.jffs2
  /dev/mtd1 && flashcp /tmp/uImage /dev/mtd2
  && echo > /dev/watchdog
```

Listing 4.7: Flashen des Dateisystems

Über den Bootloader

Der Kernel oder das Dateisystem sollte nur im Bootloader geflasht werden, wenn das Betriebssystem nicht mehr funktioniert, weil das Flashen im Bootloader bis zu einer Stunde dauern kann und durch Tippfehler die ganze Prozedur von neuem beginnt. Als Erstes muss der Gumstix an einen seriellen Port angeschlossen werden. Mit dem Befehl *minicom -s* wird auf dem angeschlossenen Rechner das Konsolenprogramm gestartet, das einem Zugriff auf die Konsole des Gumstix ermöglicht. Doch zuerst müssen im Menü folgende Parameter eingestellt werden:

- Serial Device: `/dev/ttyS0`
- Bps/Par/Bits: 115200 8N1
- Hardware Flow Control: No
- Software Flow Control: No

Nachdem die Daten gespeichert sind und die Konfiguration verlassen wurde, muss die Gumstix-Konsole mit *minicom -o* gestartet werden. Um in den Bootloader zu kommen, muss sofort nach dem Einschalten des Gumstix eine Taste gedrückt werden. Wenn ein “Gum>” erscheint, hat der Vorgang geklappt.

Zuerst muss das Dateisystems geflasht werden indem das Image des Dateisystems auf den Gumstix kopiert wird. Hierfür wird zunächst das Empfangsprogramm des Bootloaders mit

dem Befehl *loady a2000000* gestartet. Dieser Befehl sorgt dafür, dass die über den seriellen Port empfangenen Daten im Speicher an die angegebene Adresse, welche sich im RAM befindet, geschrieben werden. Als nächstes müssen natürlich auch die Daten an den Gumstix gesendet werden. Dies geschieht mittels einer Funktionalität in Minicom. Mit [CTRL]-[A] kommen wird der Menümodus geöffnet. Durch Drücken der Taste [S] wird in das Kommunikationsmenü gewechselt. Hier “yModem” auswählen. In dem nun auftauchenden Dialog das Dateisystem-Image auswählen, welches durch Kompilieren von Build-Root entstanden ist. Es trägt normalerweise den Namen “rootfs.arm_nofpu.jffs2”.

Wenn der Kopiervorgang beendet ist, muss der Flash-Vorgang mit dem Befehl *pro on 1:0-1 00 jera all 00 cp.b a2000000 40000 \${filesize}* gestartet werden. Um den Kernel zu kopieren, wird wieder das Empfangsprogramm des Bootloaders mit *loady a2000000* gestartet und mittels [CTRL]-[A] in das Menü des Minicom gewechselt, mit [S] in den Kommunikationsmodus, “yModem”, und im auftauchenden Dialog das Kernel-Image ausgewählt. Es trägt normalerweise den Namen “uImage”. Nach dem Kopiervorgang muss zum Flashen des Kernel die Befehle aus Listing 4.8 eingetippt werden. Der Kernel und das Dateisystem sollten jetzt geflasht sein. Um U-Boot zu beenden muss *bootm* eingetippt oder den Gumstix vom Strom getrennt werden.

```
# katinstall 100000
# katload 100000
```

Listing 4.8: Flashen des Kernel

4.1.6 Eigene Programme

Es gibt 2 Möglichkeiten eigene Programme auf dem Gumstix laufen zu lassen. Die erste Möglichkeit ist im Buildroot den gcc und g++ compiler hinzu zu fügen und die Programme auf dem Gumstix zu kompilieren. Die zweite Lösung wäre ein Cross-Compiling. Dazu wird im gleichen Verzeichnis, in dem sich der Ordner “gumstix-buildroot” befindet, ein neuer Ordner angelegt und die Quelldateien des Programms in diesem Ordner eingefügt. Als Nächstes muss ein Makefile angelegt werden, welcher den Inhalt aus Listing 4.9 enthält. In der ersten Zeile müssen noch die Quelldateien hinzugefügt werden, welche jeweils durch ein Leerzeichen getrennt sind. In die dritte Zeile wird Namen der gewünschten Datei geschrieben. Sollte der eigene Quellcode komplexer sein und müssen Bibliotheken hinzugefügt oder Include-Pfade gesetzt werden, so wird das Makefile um die entsprechenden Parameter erweitert. Wenn alles stimmt, kann der Kompiliervorgang mittels des Befehls *make* gestartet werden. Es kann sein, dass zu Beginn ein paar Fehlermeldungen auftauchen, dass bestimmte .o und/oder .bin-Dateien nicht gefunden werden konnten. Dies könnte darin liegen, dass zur Sicherheit bei jedem Kompiliervorgang alle .o und .bin-Dateien gelöscht werden. Es kann bei minimalen Änderungen vorkommen, dass der Cross-Compiler keine Änderungen in den Quelltextdateien erkennt und nicht kompiliert. Dies wird durch das Entfernen der .o und .bin-Dateien erzwungen. Die so entstandene Datei muss anschließend nur noch auf den Gumstix kopiert werden.

```
SOURCES =
OBJECTS = $(SOURCES:.cpp=.o)
BINARY =

all : clean $(BINARY)

$(BINARY) : $(OBJECTS)
    ../gumstix-buildroot/build_arm_nofpu/staging_dir/
    bin/arm-linux-uclibcgnueabi-g++ -o $(BINARY) $(OBJECTS)

$(OBJECTS) : $(SOURCES)
    ../gumstix-buildroot/build_arm_nofpu/staging_dir/
    bin/arm-linux-uclibcgnueabi-g++ -c $(SOURCES)

clean :
    rm $(OBJECTS) $(BINARY)
```

Listing 4.9: Makefile

4.1.7 Zusätzliche Hardware

Es werden beim Raptor verschiedene Hardware Komponenten, wie zum Beispiel Ultraschallsensoren, Kompass und Radencoder verwendet. Diese werden mittels des ATmega Mikrocontrollers (siehe Kapitel 3.2) eingebunden und angesteuert. Doch andere Hardwarekomponenten, wie der WLAN-Stick oder die WebCam verfügen über eine USB-Schnittstelle, so dass sie direkt am Gumstix angeschlossen werden können. Anhand des Beispiels einer WebCam wird gezeigt, wie Hardware direkt am Gumstix eingebunden und angesprochen werden kann.

Als erstes werden die richtigen Linux-Treiber für die WebCam benötigt. Der Raptor hat eine Logitech Quickcam Express, also wird der Treiber “GSPCA/SPCA5xx” als Quellcode benötigt.

Zuerst muss, wie in Abschnitt 4.1.6 beschrieben, der Treiber cross-kompiliert werden. Hierfür wird im gumstix-buildroot Verzeichnis ein Ordner angelegt und der Treiber in den Ordner eingefügt. Für die Videofunktionen ist wichtig, dass die entsprechenden Bibliotheken auf dem Entwicklungsrechner installiert sind. In diesem Fall muss die Bibliothek V4L (Video for Linux) vorhanden sein.

Um das cross-compiling erfolgreich durchzuführen, muss noch eine Änderung im Makefile des Treibers vorgenommen werden. Die Zeile

```
KERNELDIR := /lib/modules/$(KERNEL_VERSION)/build
```

Listing 4.10: Makefile

muss durch folgende Zeilen ersetzt werden:

```
KERNELDIR = ../gumstix-buildroot/build_arm_nofpu/  
    linux-2.6.21gum  
BUILD_ARM = ../gumstix-buildroot/build_arm\_nofpu  
CROSS_COMPILE = $(BUILD_ARM)/staging_dir/bin/arm-linux-
```

Listing 4.11: Makefile

Jetzt muss die Zeile

```
$(MAKE) -C $(KERNELDIR) SUBDIRS=$(PWD) CC=$(CC) modules
```

Listing 4.12: Makefile

durch

```
$(MAKE) -C $(KERNELDIR) SUBDIRS=$(PWD) ARCH=arm  
    CROSS_COMPILE=$(CROSS_COMPILE) modules
```

Listing 4.13: Makefile

ersetzt werden. Dadurch bekommt der Compiler die Informationen, wo sich der Kernel befindet und um welche Architektur es sich handelt. Erneut kann durch den *make* Befehl in einer Konsole der Quellcode kompiliert werden. Es entsteht eine *.ko* Datei, in diesem Fall das Kernel-Modul “gspca.ko”. Diese und die 3 zusätzlich in Listing 4.14 aufgeführten Dateien müssen nun auf den Gumstix kopiert werden, und zwar ins Verzeichnis */lib/modules/2.6.21gum/kernel/drivers/media/video/*. Zum Schluss muss in der Datei */etc/modules* der Name des neuen Moduls hinzugefügt werden, erkennbar in Listing 4.15. Nach einem Neustart des Gumstix sollte die WebCam nun einsatzbereit sein.

```
gspca.ko  
../gumstix-buildroot/build\_arm\_nofpu/root/lib/modules/  
    2.6.21gum/kernel/drivers/media/video/videodev.ko  
../gumstix-buildroot/build\_arm\_nofpu/linux-2.6.21gum/  
    drivers/media/video/v4l1-compat.ko  
../gumstix-buildroot/build\_arm\_nofpu/linux-2.6.21gum/  
    drivers/media/video/v4l2-common.ko
```

Listing 4.14: 'Cross-kompilierte Kernel Module'

```
# WebCam  
gspca
```

Listing 4.15: '/etc/modules'

4.1.8 I2C auf dem Gumstix

Der Gumstix verdex XL6P besitzt keinen separaten Anschluss für den *I²C*-Bus. Es existieren aber dennoch zwei Möglichkeiten eine Anbindung an diesen zu ermöglichen. Zum

einen ist auf dem Motherboard ein Anschluss für ein 24-poliges Folienkabel vorhanden, welches die I^2C -Anschlüsse liefert. Zum anderen kann auf dem Carrier-Board an vorgesehenen Lötstellen ein Anschluss für den I^2C -Bus angebracht werden. Es kam die zweite Variante zum Einsatz, zu sehen in Abbildung 4.1. Die Belegung des I^2C -Anschlüsse stehen in den Board-Layouts des console-vx[27].

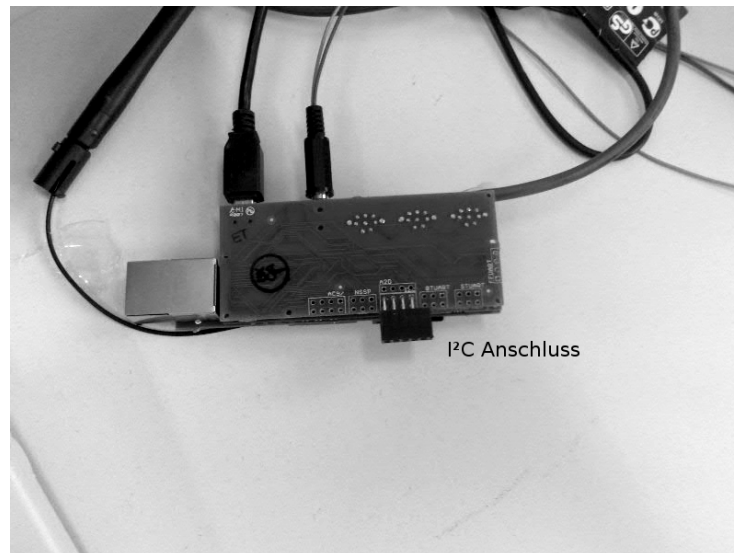


Abbildung 4.1: Anschluss an das Carrier Board

Das Kabel (Abbildung 4.2) musste von der Raptor-Gruppe selbst gebaut werden. Es wurde so entworfen, um eine falsche Benutzung zu vermeiden. Dennoch ist hier große Vorsicht geboten.

Auf dem Kabel sind farbige Markierungen angebracht. Diese kennzeichnen jede Ader individuell und unverwechselbar.

- blau = X-SCL
- gelb = X-SDA
- schwarz = GND
- weiß = VCC

Um sicher zu stellen, dass auch das I^2C -Modul im Kernel eingebunden ist (was beim Verdex Board der Fall sein sollte), können mit dem Befehl *lsmod* alle eingebundenen Module angezeigt werden. Sollte das Modul “i2c-pxa” oder “i2c-dev” nicht dabei sein, so muss in der Buildroot Konfiguration der I^2C Support aktiviert werden, der Kernel erneut kompiliert und der neue Kernel geflasht werden. Anschließend müssen in der Datei `/etc/modules` folgende Zeilen hinzugefügt werden:

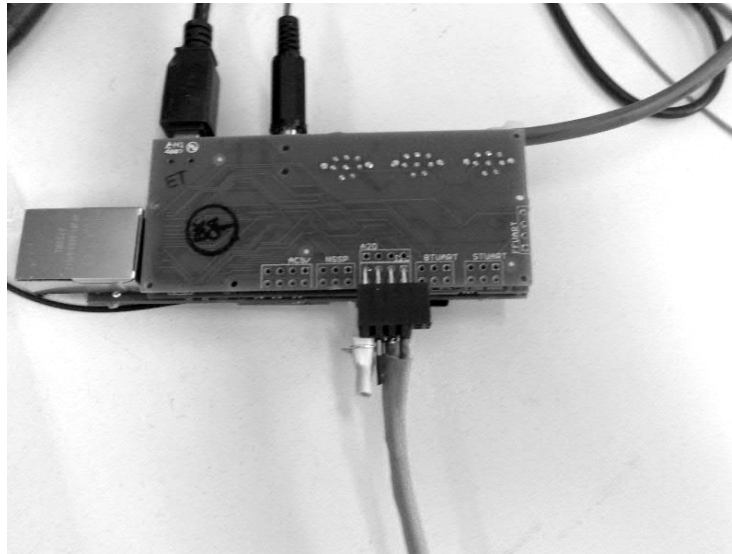


Abbildung 4.2: Carrier Board mit Anschlusskabel

```
# I2C
i2c-pxa
i2c-dev
```

Listing 4.16: /etc/modules

Nach einem Reboot sollte nun I^2C einsatzbereit sein. In der Ausgabe von *lsmod* ist zu sehen, ob die I^2C Module auch eingebunden sind.

4.1.9 I2C Kommunikation

Zu Testzwecken wird ein simples I^2C Programm[13] benutzt. Am Beispiel des Kompassensors soll die Kommunikation mittels I^2C verdeutlicht werden.

Der Kompassensor wird zunächst mit dem I^2C -Kabel verbunden. Die Adresse des Kompassensors ist "96". Um vom Kompassensor einen Wert zu erhalten, muss zunächst der Sensor in einen Zustand des Sendens überführt werden, indem ihm ein Integer (in diesem alle eine "2") als Byte geschickt. Dies erfolgt durch den ersten Befehl aus Listing 4.1.9. An die Adresse 96 wird nun ein Byte mit dem Wert 2 geschickt. Die vom Sensor geschickten Daten können mit dem zweiten Befehl aus Listing 4.1.9 empfangen werden. "`-count=2`" bedeutet, dass 2 Byte von der Adresse 96 empfangen werden sollen. Es erscheint in der Konsole nun ein Kompasswert in Hexadezimal. Um eine API für die Hardware des Raptors zu erstellen, können die Funktionen im Quellcode vom I^2C genutzt werden.

Für die Kommunikation mit dem Mikrocontroller muss zuerst ein Protokoll erstellt werden, damit sowohl der Gumstix als auch der Mikrocontroller wissen, welche Daten an welche Hardware gesendet werden sollen oder welche Daten von der Hardware zum Gumstix

geschickt werden sollen.

```
# i2c 96 SendByte 2
# i2c --count=2 96 RcvByte
```

4.1.10 Buildroot

Das Buildroot [28] ist ein System, welches ein standardisiertes Dateisystem Image basierend auf Linux für den Gumstix erstellt. Es werden alle Schritte automatisiert, die zur Erstellung eines brauchbaren Komplettsystems benötigt werden, im Einzelnen wird folgendes durchgeführt :

- Herunterladen aller nötigen Quellcodes und Daten,
- Bauen eines Cross Compilers mit der nötigen Toolchain,
- Kompilieren der gesamten Software, inklusive Kernel und Anwendungen,
- Zusammenführung aller Ergebnisse in ein fertiges Dateisystem, das auf den Gumstix Flash Speicher geladen werden kann.

Im Folgendem wird genauer darauf eingegangen, wie ein Dateisystem Image erstellt wird und wie der Cross Compiler zu benutzen ist.

Kompilieren des Buildroots

Um ein Buildroot zu erstellen, müssen folgende Programme vorhanden sein:

- Subversion,
- GNU Compiler gcc,
- autoconf.

Zunächst wird der aktuelle Quellcode mit Hilfe von Subversion vom Gumstix-Server heruntergeladen. Dabei ist zu beachten, dass, falls ein Proxy benutzt wird, dieser in die Umgebungsvariablen `http_proxy` und `ftp_proxy` des Systems eingetragen werden. Jetzt kann mit folgendem Befehl ein check out ausgeführt werden :

```
svn co http://svn.gumstix.com/gumstix - buildroot/trunk gumstix - buildroot.
```

Dabei wird der Quellcode in das Verzeichnis `gumstix-buildroot` kopiert. Jetzt wird in das `gumstix-buildroot` Verzeichnis gewechselt und das Buildroot kann konfiguriert werden. Dafür gibt es zwei Möglichkeiten.

Die erste Möglichkeit ist eine schnelle Lösung. Dazu wird der Befehl

```
make defconfig
```

eingegeben. In unserem Fall wird bei der “CPU Speed“ die ”600 Mhz“ Option gewählt und die restlichen Fragen werden mit der vorgegebenen Option beantwortet.

Die zweite Möglichkeit ist sehr viel detaillierter. Dazu wird der Befehl

make menuconfig

benötigt. Dabei wird ein Menü geöffnet, wo das zu erstellende Image sehr genau konfiguriert werden kann. Es kann zum Beispiel aus einer großen Anzahl von Programmen ausgewählt werden, welche für den Gumstix bereitgestellt werden sollen und welche nicht.

Nachdem die Konfigurierung abgeschlossen ist, kann die Erstellung des Images durch den Befehl

make

gestartet werden. Nach ungefähr einer Stunde ist das Image im Verzeichnis zu finden. Jetzt kann dieses auf den Gumstix übertragen werden.

Cross Compiler

Ein Cross Compiler ist ein Compiler, der auf einem Hostsystem (in unserem Fall Linux) installiert ist und Quellcode für eine andere Zielplattform (in unserem Fall ARM) übersetzt, so dass das die entstanden Programmdateien auf der Zielplattform ausgeführt werden können.

Der vom Buildroot erstellte Cross Compiler ist für die meiste Software zu gebrauchen, außer natürlich für Software die speziell für eine Architektur geschrieben wurde, die nicht ARM entspricht.

Die meisten Programme lassen sich mit Hilfe von “autoconf” automatisch konfigurieren. Um einen Quellcode zu übersetzen, wird in das Hauptverzeichnis des Quellcodes gewechselt und durch den Befehl

./configure --host = arm - linux

autoconf gestartet. Dabei ist zu beachten, dass sich ein Verweis auf den Cross Compiler in der PATH Variable befinden muss. Ist das nicht der Fall, muss diese gesetzt werden. Jetzt kann der Übersetzungsvorgang durch den Befehl

make

gestartet werden. Wenn die Übersetzung ohne Fehler abgelaufen ist, kann das Programm auf den Gumstix übertragen und ausgeführt werden.

4.2 Gitterkarte

Die Gitterkarte stellt die interne Repräsentation der Welt dar. In ihr werden alle relevanten Informationen gespeichert, welche es unserem Roboter ermöglichen sich in der Umwelt

zurecht zu finden. Zudem wurden ein paar höhere Funktionen in ihr gekapselt, um an dieser Stelle Performanzvorteile erzielen zu können.

Die Gitterkarte wurde komplett neu entworfen und implementiert. Dies hatte zum Ziel eine möglichst schnelle Ansprechzeit zu erreichen. Da es sich bei der Kartenrepräsentation um das Kernstück der datenverarbeitenden Funktionen unseres Roboters handelt, wurde hierbei Wert auf robuste Techniken zur Datenhaltung gelegt, welche einen korrekten Betrieb erlauben. Es wurde kein fremder Code benutzt, um schwer zu findende Fehlerquellen auszuschließen. Auch das höhere Verständnis der internen Vorgänge war ein Kriterium, welches zu der Entscheidung der Neuimplementierung geführt hat. Dies ermöglicht es einen hohen Grad der Optimierung zu erreichen, welcher bei der begrenzten Rechenleistung unseres Roboters nötig ist.

Die Gitterkarte besteht insgesamt aus drei Klassen, welche im Weiteren näher beleuchtet werden.

4.2.1 GridMap

Begonnen wird mit der Klasse, die die Schnittstelle der Gitterkarte zum Rest der Software darstellt. Sie kapselt zum einen die eigentliche Datenverwaltung gegenüber dem Zugriff von außen ab und stellt zudem höhere Funktionen bereit, die verschiedene Aufgaben implementieren. Zunächst wird kurz auf die eigentliche Datenhaltung eingegangen. Diese ist implementierungsbedingt nicht Teil der Klasse GridMap. Es wird aber durch diese Klasse der Zugriff auf den Datenspeicher ermöglicht. Dieser wird durch die Klasse GridMapData implementiert, auf welche später eingegangen wird. Die Funktionen, die von außen zugänglich sein müssen, sind zu aller erst die Get-/Set-Methoden, welche den Zugriff auf die grundlegenden Eigenschaften der einzelnen Gitterkartenzellen ermöglichen. Dies sind die Belegungswahrscheinlichkeit und der Verschmutzungsgrad. Näheres hierzu wird bei der Besprechung der Klasse Feld dargestellt. Es reicht erst einmal zu wissen, dass hierbei die entgegengenommenen Anfragen an die Klasse GridMapData weitergeleitet werden, ohne dass eine Veränderung der zu transferierenden Daten stattfindet. Des Weiteren ist noch die räumliche Ausdehnung der Gitterkarte verfügbar. Diese Werte werden auf Anfrage bei der Klasse GridMapData angefordert, welche diese während der Laufzeit ständig aktualisiert. Zu Letzt ist noch die sogenannte "Sondereigenschaft" der Klasse Feld (ein temporärer Datenspeicher) über Get-/Set-Methoden zu erreichen. Neben den gerade aufgeführten Funktionen, die die Datenhaltung koordinieren stehen in der Klasse GridMap noch weitere datenverarbeitende Funktionen zur Verfügung. Diese bilden den eigentlichen Sinn der Trennung zwischen den Klassen GridMap und GridMapData. Diese Funktionen bieten drei Hauptfunktionalitäten, welche im Folgenden näher dargestellt werden.

Laden-Speichern

Naturgemäß liegt die aktuelle Karte im schnellen Hauptspeicher vor. Da dieser aber beim Ausschalten gelöscht wird, müssen die erfassten oder berechneten Informationen der Gitterkarte auf einem nichtflüchtigen Speicher gesichert werden. Hierfür hält die Klasse GridMap

zwei Funktionen bereit.

```
void loadGridMapFromFile(string dateiname);  
void saveGridMapToFile(string dateiname);
```

Listing 4.17: Laden-Speichern

Die Funktionsnamen sind hierbei fast selbsterklärend. Die Methode `loadGridMapFromFile` lädt aus der angegebenen Datei eine Gitterkarte ein und `saveGridMapToFile` sichert die aktuelle Karte in der Datei mit dem angegebenen Dateinamen. Was sich nun sehr simpel anhört ist jedoch in der Implementierung etwas anspruchsvoller. Es wurde ein neues Dateiformat entwickelt, welches darauf ausgelegt ist eine effiziente Speicherung zu erzielen. Die Effizienz bezieht sich hierbei vor allem auf die zum Laden/Speichern notwendige Rechenleistung. Da auf einem Embedded Computer gearbeitet wird, ist dieser Faktor von enormer Bedeutung. Die Speicherplatzfrage stellt durch die geringen Kosten für zusätzlichen Speicher kein Problem dar. Daher wurde ein unkomprimiertes Format entworfen. Der zusätzlich nötige Overhead konnte auf 4 Byte reduziert werden. Diese 4 Byte stehen am Anfang einer jeden gespeicherten Gitterkarte. Sie geben an, welche Breite die Gitterkarte aufweist. Alle sonst benötigten Informationen werden während des Einlesevorgangs ermittelt. Die Höhe der Gitterkarte ergibt sich beispielsweise aus der Breite sowie der Anzahl der Daten in der Datei. Es werden also einfach solange komplette Zeilen eingelesen, bis keine Daten mehr in der Datei zu finden sind. Beim Speichern wiederum schreibt man solange ganze Zeilen, bis die Gitterkarte komplett erfasst wurde. Das Format der einzelnen Zellen ist hierbei auch auf das Wesentliche reduziert. Gerade einmal 2 Byte müssen pro Feld gespeichert werden. Das erste Byte enthält die Belegungswahrscheinlichkeit und das zweite den Verschmutzungsgrad. Die Wertebereiche dieser beiden Eigenschaften liegen im Bereich 0 bis 100. Sollten sie dennoch in der Datei den Wert 255 aufweisen, symbolisiert dies, dass dieses Feld noch nicht erforscht wurde und somit leer ist.

Objekterkennung

Es folgt nun eine Beschreibung der beiden komplexen Aufgaben der Klasse `GridMap`. Diese sind, auf der Karte einzelne Objekt zu identifizieren und lokalisieren, sowie Gegenstände als Ganzes zu erkennen. Hierfür werden zwei Funktionen zur Verfügung gestellt.

```
vector< vector<Feld> > getSegmentVector();  
vector< vector<Feld> > getSegmentRandVector();
```

Listing 4.18: Objekterkennung

Die Funktion `getSegmentVector` ermittelt mit Hilfe von Techniken der Bildverarbeitung die einzelnen Objekte auf der Karte und gibt diese gesammelt zurück. Hierbei wird die Datenstruktur "Vector" aus der Standard Template Library [24] verwendet, welche eine Menge von Objekten einer beliebigen Datenstruktur enthalten kann. Der äußere "Vector" repräsentiert hierbei die Menge der Objekte und die inneren "Vektoren" die einzelnen Objekte, welche wiederum aus den Feldern bestehen, die durch dieses Hindernis belegt werden.

Näheres zur Datenstruktur “Vector” wird bei der Erläuterung der Datenstruktur beleuchtet, welche die Datenhaltung der Gitterkarte implementiert. An dieser Stelle bleibt nur festzuhalten, dass durch die Funktion `getSegmentVector` alle Felder zurückgegeben werden, welche von Objekten belegt werden. Diese sind nach den einzelnen Objekten sortiert. Es bleibt nun noch die Frage zu klären, wie diese Daten ermittelt werden. Hierfür kommt ein Algorithmus zum Einsatz, welcher sich aus zwei Segmentierungsalgorithmen der Bildverarbeitung ableitet. Diese sind zum einen das Schwellwertverfahren [29] und zum anderen das Bereichswachstumsverfahren [17]. Der grobe Ablauf lässt sich mit folgendem Algorithmus beschreiben:

```
for (alle Felder) do
    if (Feld belegt) und (Feld keinem Objekt zugeordnet)
        lasse Objekt wachsen
    end;
end;
```

Listing 4.19: Objekterkennungsalgorithmus

Die aufgeführte If-Bedingung ist aus dem Schwellwertverfahren [29] abgeleitet. Hierbei wird anhand eines vorher definierten Wertes festgelegt, ab welcher Belegungswahrscheinlichkeit ein Feld als belegt angenommen wird. Die Zeile “lasse Objekt wachsen” realisiert das Bereichswachstumsverfahren [17]. Hierbei wird iterativ versucht in alle Richtungen zu “wachsen”. Dieses Vorgehen ist in Abbildung 4.3 dargestellt. Die Zugehörigkeit eines Feldes zu einem Objekt wird über die Eigenschaft `m.segment` in der Klasse `Feld` gesichert. Eine weitere Eigenschaft die in diesem Arbeitsschritt ermittelt wird, ist die Eigenschaft eines Randpixels (da die verwendeten Algorithmen aus der Bildverarbeitung stammen, sind hier der Begriff des Pixels und des Feldes äquivalent). Ein Feld ist ein Randpixel, wenn er nicht von allen Seiten von Feldern des gleichen Objektes eingerahmt ist. Wird dies festgestellt, dann wird die Eigenschaft `m.randpixel` in der Klasse `Feld` entsprechend gesetzt. Dies wird später noch von Bedeutung sein.

Nachdem nun die erste der beiden Objekterkennungsfunktionen besprochen wurde, wird im folgenden die Zweite erläutert. Die Funktion `getSegmentRandVector` wurde ursprünglich implementiert, um die Eingabe für die zu berechnende Voronoi-Karte zu ermitteln. Da diese aus verschiedenen Gründen verworfen wurde, ist die Funktion `getSegmentRandVector` nicht mehr von großer Bedeutung. Es kann jedoch sein, dass sie zur Wiedererkennung der Räume benötigt wird, um dort einen Performanzvorteil zu erreichen. Die Rückgabe der Funktion ähnelt stark der Rückgabe der Funktion `getSegmentVector`, wobei hier nicht die gesamten Objekte zurückgegeben werden, sondern nur die Randpixel in benachbarter Reihenfolge. Dies bedeutet, dass auf einen Randpixel direkt ein Nachbar-Randpixel in dem inneren “Vector” folgt. Dies war wichtig um die äußere Struktur eines Objektes analysieren zu können. Die Anforderung zeigt, warum diese benötigten Werte nicht innerhalb der Funktion `getSegmentVector` ermittelt wurden. Dort werden die Randpixel in zufälliger Reihenfolge gefunden, was nicht akzeptabel war. Also musste folgender Algorithmus umgesetzt werden:

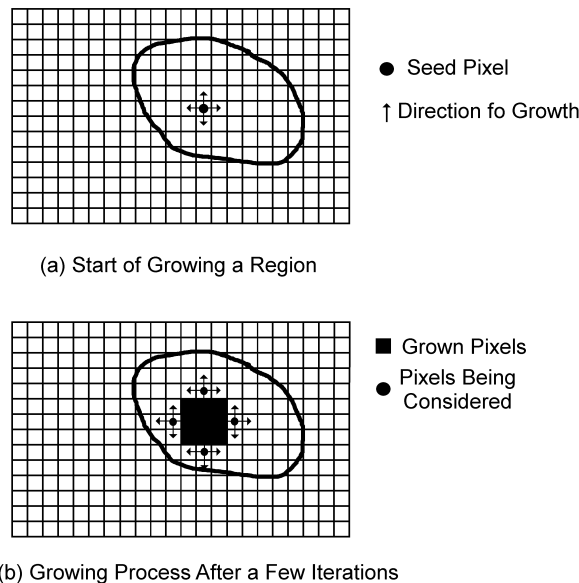


Abbildung 4.3: Bereichswachstumsverfahren [17]

```
for (alle Felder in zeilenweiser Reihenfolge) do  
  if (Feld ist Randpixel)  
    ermittel Rand  
  end;  
end;
```

Listing 4.20: Wachstumsalgorithmus der Objekterkennung

Dies führt erst einmal dazu, dass jeder Rand eines Objektes überhaupt gefunden wird. Die eigentliche Aufgabe bestand darin, den einzelnen Rand geschickt zu ermitteln, so dass er in geordneter Reihenfolge gespeichert werden kann. Hierzu wurde folgende Schleife umgesetzt:

```
while (aktuelles Feld != Startfeld)  
  finde Nachbarrandpixel  
  aktuelles Feld der Rückgabe hinzufügen  
  setze Nachbarrandpixel als aktuelles Feld  
end;
```

Listing 4.21: Randerkennung

Dies führt aufgrund der spezifischen Eigenschaften eines Objektrandes dazu, dass die Randpixel in ihrer Nachbarschaftsreihenfolge gefunden werden. Diese Eigenschaften sind, dass ein Feld, welches Teil eines Objektrandes ist, genau zwei Nachbarn besitzt, die ebenfalls Teil des gleichen Objektrandes sind. Im Verlauf einer Suche, ist das aktuell betrachtete

Feld dadurch gekennzeichnet, dass ein Objektrandnachbar bereits gefunden wurde, aber der Zweite noch nicht. Diese Tatsache ist nur bei dem ersten und letzten Feld eines Objektrandes nicht gegeben.

Routenplanung

Es folgt nun die Routenplanung. Diese dient dazu einen möglichst optimalen Weg von einem angegebenen Startpunkt zu einem gewünschten Zielpunkt zu berechnen. Der Grund, warum diese Funktionalität innerhalb des datenverarbeitenden Teil der Gitterkarte gekapselt ist, liegt bei näherer Betrachtung auf der Hand. Es wird ein Algorithmus verwendet, der direkt auf der Gitterkarte arbeitet und eine speziell für solche Zwecke implementierte Sondereigenschaft der einzelnen Felder nutzt. Diese Sondereigenschaft werden wir näher bei Besprechung der Klasse Feld betrachten. Es handelt sich hierbei um eine einfache Variable in den Feldern, welche höhere Funktionen bei Berechnungen unterstützen und entlasten soll. In dem Fall der Routenplanung werden hier Schätzwerte der Entfernungen vom Startpunkt gespeichert und anschließend ausgewertet. Bevor mit der detaillierten Beschreibung dieser Funktion begonnen wird, werden noch ein paar Vorüberlegungen zur Routenberechnung angeführt. Wie bereits zu einem früheren Zeitpunkt erwähnt, ist die Gitterkarte intern unabhängig von der Größe der einzelnen Felder. Dennoch müssen ihr diese Angaben mitgeteilt werden. Auch die Größe des Roboters ist hier von Bedeutung. Es muss ein gewisser Abstand zu den Objekten im Raum eingehalten werden. Um diesen zu berechnen, werden die gerade erwähnten Werte gebraucht.

```
static const int FELD_BREITE;  
static const int ROBOTER_DURCHMESSER;
```

Listing 4.22: Konstanten für die Routenplanung

Konstanten tragen der Tatsache Rechnung, dass es Werte gibt, die sich nicht ändern. Hier liegen zwei solcher Beispiele vor. Der Roboter wird nicht plötzlich schrumpfen und die Räume sich auch nicht ausdehnen. Daher können mit Hilfe der beiden obigen Konstanten die Berechnungen zur Routenplanung durchgeführt werden. Hierfür muss aus den beiden Konstanten die Anzahl der freizuhaltenden Felder berechnet werden, die den Abstand vom Zentrum des Roboters zu den Objekten im Raum darstellen. Ist dies geschehen, kann dies bei der Schätzung der Entfernungen berücksichtigt werden. Für diesen Vorgang wird folgender Algorithmus umgesetzt, der sich an dem Bereichswachstumsverfahren[17] orientiert.

```
weise Startfeld die Entfernung 1 zu  
lege Startfeld auf Stack  
for (alle Felder auf dem Stack) do  
    hole TOS -> aktuelles Feld  
    for (alle Nachbarn des aktuellen Feldes) do  
        wenn (Nachbarfeld nicht belegt)  
            and (Nachbarfeld nicht zu nah an Objekt) do
```

```
        weise Nachbarfeld sein Entfernung zu
        lege Nachbarfeld auf Stack
    end
end
end
```

Listing 4.23: Entfernungsschätzung

Die Entfernungsschätzung kann hier auf vielfältige Art und Weise erledigt werden. Es wurde dabei Wert auf die Verwendung von simplen und schnellen Berechnungen gelegt. Als Nachbarn werden die acht Nachbarfelder und nicht wie beim Bereichswachstumsverfahren nur die vier Nachbarn in den Himmelsrichtungen betrachtet. Die Nachbarn in den vier Himmelsrichtungen bekommen hierbei die Entfernung des aktuellen Feldes + 1. Die anderen vier Nachbarn würden die Entfernung des aktuellen Feldes + $\sqrt{2}$ erhalten. Da diese Präzision aber nicht nötig ist wird hier auf die Wurzelberechnung verzichtet und die Entfernung des aktuellen Feldes + 1.414 angenommen. In Abbildung 4.4 kann ein Beispiel für diese Schätzung betrachtet werden. Die Entfernungswerte wurden auf der Darstellung gerundet und modulo 10 gerechnet, um auf der Konsole eine verwertbare Darstellung zu erreichen. Mit etwas Überlegung kann schnell der Startpunkt der Routenplanung erkannt werden. Er befindet sich direkt unter dem linken unteren Hindernis auf dem Feld, welches mit einer 1 geschätzt wurde. Der zusätzlich eingehaltene Abstand zu den Objekten und Kartenrändern kann ebenfalls wahrgenommen werden. Letzteres resultiert aus der Tatsache, dass unbekannte Regionen potentiell Hindernisse enthalten können und daher ein Sicherheitsabstand eingeplant werden muss. Nun muss jedoch aus den berechneten Entfernungen die eigentliche Route extrahiert werden. Hierfür wird am angegebenen Zielpunkt gestartet und der folgende Algorithmus umgesetzt:

```
    lege Zielpunkt auf Stack
    setze Zielpunkt als aktuelles Feld
    while (aktuelles Feld != Startpunkt) do
        for (alle Nachbarn vom aktuellen Feld) do
            suche kürzeste Entfernung
        end
        lege Nachbarn mit kürzester Entfernung auf Stack
        Nachbar mit kürzester Entfernung wird aktuelles Feld
    end
    return stack
```

Listing 4.24: Routenextraktion

Hierbei werden als Entfernungen die zuvor berechneten Entfernungen vom Startpunkt verwendet. Die Interpretation könnte sein, dass Entfernungen als Höhenwerte eines Gebirges angenommen werden. Am Zielpunkt entspringt ein Fluss und der Weg den das Wasser zum Startpunkt wählt ist die kürzeste Route. Hierbei kann es auch mehrere Routen geben, was dazu führt, dass bei der Suche nach dem Nachbarn mit der kürzesten Entfernung

mehrere mögliche Kandidaten gefunden werden. In diesem Fall wählt man beliebig aus, denn die Länge der Route wird auf jeden Fall minimal sein. Wie die kürzeste Route in dem aufgeführten Beispiel aussieht ist in Abbildung 4.5 zu sehen. Die Zahlen in der Abbildung 4.5 stellen hier nicht die zuerst berechneten Entfernungen dar, sondern die Reihenfolge der Felder modulo 10. Durch den verwendeten Stack werden also nicht nur die Felder auf der Route ermittelt, sondern auch die Reihenfolge in der diese angefahren werden müssen.

Dieses ganze Verfahren hat sich als schnell herausgestellt. Tests auf der von uns verwendeten Hardware zeigten auch bei größeren Karten gute Leistungsergebnisse. Trotzdem können noch mehrere Optimierungen eingebaut werden. Zum Beispiel kann die resultierende Route auf wenige markante Punkte reduziert und mit verschiedenen Verfahren aus der Bildverarbeitung eine glattere Kurve berechnet werden, die nicht dazu führt, dass sich der RAPTOR auf der Stelle drehen muss, wie es bei einer eckigen Route der Fall ist. Hier muss sich aber noch zeigen, dass der Rechenaufwand an dieser Stelle gerechtfertigt ist. Eine weitere Optimierung kann während der Schätzung der Entfernungen erreicht werden, indem nur vielversprechende Felder ausgewertet werden, die mit einer hohen Wahrscheinlichkeit die optimale Route beschreiben. Dies ist aber wieder eine Optimierung, die gerechtfertigt sein muss. Es kann durchaus sein, dass die Performanz der durchgeführten Berechnungen nicht weiter ins Gewicht fällt und die Arbeit an anderer Stelle sinnvoller ist.

4.2.2 GridMapData

Die funktionale Trennung zwischen Datenhaltung und Datenverarbeitung resultiert neben der Klasse GridMap in der Klasse GridMapData. Hier wird jegliches Wissen über die Welt abgelegt und für die Auswertung bereitgehalten. Es werden hier aber keinerlei datenverarbeitende Funktionen bereitgestellt.

Die von außen aufrufbaren Funktionen sind fast alle Get-/Set-Funktionen, die bestimmte Daten an einer bestimmten Stelle der Karte modifizieren. Hierbei wird in einem ersten Schritt das entsprechende Feld der Gitterkarte adressiert und anschließend die entsprechende Funktion in dem ermittelten Feld angesprochen. Die eigentlichen Daten werden nur durchgereicht und nicht verarbeitet. Dies stellt sicher, dass keine ungewollte Manipulation der Inhalte stattfindet, welche sich nur schwer auf Fehler untersuchen lassen würde. Welche Daten genau manipuliert werden können, wird bei der Beschreibung der Klasse Feld erläutert. Alle dort zu findenden öffentlichen Funktionen resultieren in einer entsprechenden Funktion in der Klasse GridMapData. Der Aufruf unterscheidet sich nur dadurch, dass zusätzlich die Koordinaten des entsprechenden Feldes angegeben werden müssen.

Neben diesen Funktionen werden noch ein paar andere Daten vorgehalten. Diese beziehen sich nicht auf die Daten, welche in der Karte eingetragen werden. Es handelt sich hierbei um Informationen, die eine schnellere Auswertung der Karte ermöglichen. Es handelt sich um die Ausdehnung der Karte in X- und Y-Richtung. Die Funktionen hierfür sind Folgende:

```
int getMaxY();  
int getMaxX(int y);  
int getMaxX();
```

```

int getMinY();
int getMinX(int y);
int getMinX();

```

Listing 4.25: Kartengrößenausgabe

Die Get-Funktionen ohne jegliche Parameter stellen hierbei globale Minima, bzw. Maxima dar. Die beiden Funktionen, welche einen Parameter besitzen, liefern die maximale/minimale Ausdehnung der X-Koordinaten unter der Voraussetzung der angegebenen Y-Koordinate. Die letzteren beiden Funktionen können zur späteren Laufzeitoptimierung genutzt werden, befinden sich aber noch an keiner Stelle in Gebrauch. Ein Grund hierfür ist, dass diese Daten auf Anforderung berechnet werden müssten, was ineffizient ist. Eine Überwachung dieser entsprechenden Werte kann hier Abhilfe leisten. Ob dies jedoch nötig wird, kann zu diesem Zeitpunkt noch nicht gesagt werden. Die anderen Get-Funktionen dienen der kompletten Erfassung der Karte bei Aufgaben, wie der Objekterkennung oder Ausgabe als Bitmap. Die Werte hierfür werden bereits während des Zugriffs auf die Gitterkarte ermittelt und ständig aktualisiert. Dies führt zu einem kleinen Overhead bei der Verarbeitung, welcher aber durch die drastische Beschleunigung der höheren Funktionen kompensiert wird.

Die Hauptaufgabe der Klasse GridMapData besteht darin, die zu verwaltenden Daten zu organisieren. Dies geschieht mit Hilfe der Datenstruktur “Vector[25]”, welche durch die Standard Template Library [24] zur Verfügung gestellt wird. Diese Datenstruktur dient ursprünglich dazu einen Vektor[30] zu verwalten, welcher eine beliebig hohe Dimension besitzt. Hierbei muss sichergestellt werden, dass effizient auf jeden einzelnen Eintrag zugegriffen werden kann. Ein “Vector[25]” kann also auch als dynamisches Array verstanden werden, welcher während der Laufzeit vergrößert oder verkleinert werden kann. Die Tatsache, dass die einzelnen Daten von jedem beliebigen Typ sein können - deshalb wird auch von Daten-Containern gesprochen - machen den “Vector[25]” zur idealen Datenstruktur für unsere Gitterkarte. Um nun eine zweidimensionale Karte mittels eines “Vector[25]” zu verwalten, wird ein äußerer “Vector[25]” angelegt, dessen einzelne Daten wiederum “Vektoren[25]” sind. Diese inneren “Vektoren[25]” enthalten dann die Objekte vom Typ Feld, welcher später genauer betrachtet werden. Auch wenn diese Organisationsstruktur einfach und performant ist, müssen noch ein paar Überlegungen über die exakte Umsetzung angestellt werden. In der ersten Implementierung der Klasse GridMapData wurde ein einzelner äußerer “Vector[25]” angelegt, welcher wiederum an jeder Position wieder einen einzelnen “Vector[25]” besaß (Abbildung 4.6). Hierbei kommt es jedoch zu Problemen mit der Speicherung einer Karte. Die Gitterkarte muss während der Laufzeit in alle Richtungen vergrößert, bzw. verkleinert werden können. Da jedoch der “Vector” intern einer Liste gleicht, welche nur in eine Richtung wachsen kann, beschränkt dies das Wachstum der Karte. Dies haben wir mit Hilfe der folgenden Zuordnungsfunktion gelöst:

$$x' = \begin{cases} 2*x & \text{wenn } x \geq 0 \\ (-2)*x-1, & \text{wenn } x < 0 \end{cases}$$

$$y' = \begin{cases} 2*y & \text{wenn } y \geq 0 \\ (-2)*y-1, & \text{wenn } y < 0 \end{cases}$$

Die auf diese Weise berechneten Koordinaten repräsentieren die Position innerhalb der Datenstruktur. Der äußere “Vector[25]” repräsentiert hierbei die Y-Koordinate und die inneren “Vectoren[25]” die X-Koordinaten. Die Zuordnungsfunktion bewirkt, dass der Wertebereich der Gitterkarte, welche von $-\infty$ bis ∞ geht, auf einen Wertebereich abgebildet wird, der von 0 bis ∞ geht. Hierfür wird der Zielbereich aufgeteilt in gerade und ungerade Stellen. Die geraden werden benutzt, um die ursprünglich positiven Koordinaten zu repräsentieren. Die ungeraden Bereiche nehmen die ursprünglich negativen Koordinaten auf. Die bedeutet zum Beispiel, dass die Koordinaten der Elemente eines “Vectors” auf die Weise umgesetzt werden, wie in Tabelle 4.1 zu erkennen ist.

Koordinate in “Vector”	0	1	2	3	4	5	6	7	8
Koordinate auf Karte	0	-1	1	-2	2	-3	3	-4	4

Tabelle 4.1: Beispiel für die Umrechnung von Koordinaten

Das Ganze wird sowohl bei der X- als auch Y-Koordinate gemacht. In der Klasse GridMapData wurde hierzu eine separate Funktion implementiert, die die übergebenen Koordinaten in die Zielkoordinaten umgerechnet hat. Dadurch, dass die eigentliche Datenstruktur nicht von außen zugänglich war, konnte sichergestellt werden, dass stets die richtigen Felder adressiert wurden. Jede Funktion hat dafür zu aller erst die übergebenen Koordinaten anhand der gerade erwähnten Konvertierungsfunktion umgerechnet und erst dann auf die Dateninhalte zugegriffen. Dies alles führte zu einer robusten Implementierung, die nur überarbeitet wurde, um einen Leistungsvorteil zu erreichen. Und dieser fiel größer aus, als erwartet wurde. Es zeigte sich, dass es zu einem sehr hohen Anteil an leeren Feldern in der Gitterkarte kommen würde. Dies kann ganz einfach gezeigt werden, da Karten, die nur positive Koordinaten besitzen dazu führen, dass die Gitterkarte in gleichem Maße in die negativen Bereiche des Koordinatensystems wächst. Ein “Vector” kann nämlich keinerlei Fehlstellen aufweisen. Und da zwischen zwei positiven Koordinaten intern eine negative Koordinate abgelegt ist, muss diese auch auf jeden Fall erzeugt werden.

Die Änderung der internen Repräsentation führte dazu, dass intern vier Karten verwaltet werden. Jede ist hierbei für einen Quadranten des Koordinatensystems zuständig. Da Quadranten an den Hauptachsen enden, sind sie nicht in vier Richtungen unendlich, sondern nur in zwei. Eine Kombination von äußeren und inneren “Vectoren” leistet genau dies. Wie in Abbildung 4.7 zu erkennen ist, lassen sich die Koordinaten der Gitterkarte nun intuitiv auf die interne Datenstruktur übertragen. Die Adressierung gestaltet sich nur unerheblich schwieriger. Hierfür wird in einem ersten Schritt anhand der Koordinaten der entsprechende Quadrant ermittelt, in dem die aktuellen Koordinaten zu finden sind. Dies wird durch zwei If-Bedingungen geleistet, die die Negativität der X-, bzw. Y-Koordinaten überprüfen. Die für den ermittelten Quadranten zuständige Teildatenstruktur wird nun mittels eines Pointers übergeben. Anschließend wird der Betrag der Koordinaten ermittelt und dieser im Weiteren verwendet. Der große Vorteil dieses Vorgehens ist es, dass alle

nachfolgenden Schritte durch die gleichen Funktionen realisiert werden können. Es sind also keine Unterscheidungen der vier Quadranten notwendig. Das einzige was hierbei zu bedenken ist, ist der Aufenthaltsort der Hauptachsen. Diese beinhalten alle Koordinaten in denen eine 0 vorkommt. Hier muss entschieden werden, welche “Vectoren” die 0 zugewiesen bekommen und in welchen “Vectoren” diese leer bleiben. Im Detail geht es darum wo in der Quadrantermittlung in den If-Bedingungen ein $\geq$ statt eines $>$ steht.

Der Vergleich der beiden Möglichkeiten wurde anhand mehrerer Karten vorgenommen. Dies sind zum Beispiel zufällige Karten, oder selbst konstruierte. Es wird an dieser Stelle nicht auf Einzelheiten eingegangen, da es sich nur um oberflächliche Messungen gehandelt hat. Es hat sich jedoch gezeigt, dass bei vielen sequenziellen Abfragen auf gleichförmigen (bzgl. der Ausmaße) Karten eine Leistungssteigerung von 30 Prozent realistisch ist. In der Realität treten in hohem Maße lokal begrenzte Zugriffe auf, welche empfindlicher gegenüber langsamen Zugriffen sind. Zusätzlich wird mit unförmigen Karten zu rechnen sein, welche in der ersten Version große leere Bereiche in der Karte zur Folge gehabt hätte. Dies alles führt zu der Annahme, dass ein deutlich stärkerer Leistungsgewinn in der Realität erzielt werden kann. Eine weitere Optimierung der Struktur der Gitterkarte erscheint zum gegenwärtigen Zeitpunkt nicht erforderlich. Es wäre eine zu hohe Investition an Zeit notwendig, welche nur zu einem niedrigen Leistungsgewinn führen würde.

4.2.3 Feld

Diese Klasse stellt die kleinste Einheit der Gitterkarte dar. In ihr werden die elementaren Informationen gespeichert, welche an eine bestimmte Position im Raum gekoppelt sind. Diese sind z.B. der Verschmutzungsgrad und die Belegungswahrscheinlichkeit. Zusätzlich werden hier noch Daten zwischengespeichert, welche für höhere Funktionen der Gitterkarte benötigt werden. Dies umfasst Informationen zur Objektzugehörigkeit, sowie Informationen über die Eigenschaft des einzelnen Feldes innerhalb eines größeren Objektes. Die Klasse Feld stellt hierbei nur einen Container zur Verfügung, welcher die entsprechenden Daten aufnimmt und über Get- und Set-Methoden zugänglich macht. Es existiert an dieser Stelle keinerlei datenverarbeitende Funktionalität. Auf die Angabe von Quelltext wird an dieser Stelle aufgrund der Einfachheit dieser Klasse verzichtet.

Die einzelnen gespeicherten Informationen sind Folgende:

m_belegung Diese Variable vom Typ Integer legt für das aktuelle Feld fest, mit welcher Wahrscheinlichkeit es belegt ist. Die Angabe erfolgt in Prozent, weshalb der Wertebereich zwischen 0 und 100 liegt. Die Konstante 100 bedeutet in diesem Zusammenhang, dass das aktuell betrachtete Feld mit einer Wahrscheinlichkeit von 100 Prozent belegt ist.

m_schmutz Auch bei dieser Variable handelt es sich um einen Integerwert. Er legt für das aktuelle Feld fest, wie stark der Verschmutzungsgrad ist. Hierbei wird der Wertebereich von 0 bis 100 eingegrenzt, wobei 100 die höchste mögliche Verschmutzung darstellt. Da jedoch noch nicht damit begonnen wurde, die Verschmutzung festzustellen, kann hier noch nicht von einer gefestigten Information gesprochen werden.

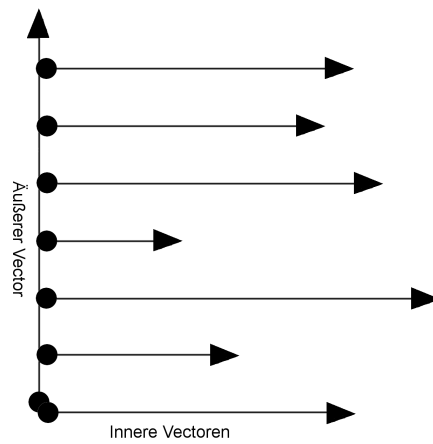


Abbildung 4.6: erste Datenstruktur zur Verwaltung der Gitterkarte

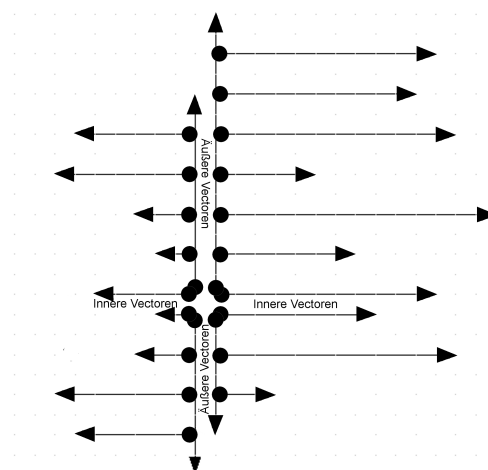


Abbildung 4.7: finale Datenstruktur zur Verwaltung der Gitterkarte

Es könnte daher sein, dass in Zukunft mehr als nur eine Variable zur Schmutzdetektion herangezogen werden muss. Zunächst wird aber davon ausgegangen, dass diese Information auf einen einzelnen Wert reduzierbar ist.

m_x, m_y Diese Werte repräsentieren die Koordinaten des aktuellen Feldes. Diese Information kann zwar auch aus der übergeordneten Datenstruktur ermittelt werden, jedoch ist es aus Performanzgründen besser, diese Information redundant abzuspeichern. Hierbei ist es nicht möglich die Koordinaten im Nachhinein zu verändern. Diese Read-Only Eigenschaft wird dadurch erreicht, dass keinerlei Funktionen zu Manipulation dieser Variablen existieren. Während der Instanziierung können über den Konstruktor die Koordinaten als Parameter angegeben werden. Ab diesem Zeitpunkt steht diese Information fest.

m_segment Dieser Wert beschreibt die Segmentzugehörigkeit. Dies ist bei der Objekterkennung auf der Gitterkarte von Bedeutung, da hier eine Segmentierung der Karte vorgenommen wird. Hierbei wird jedem alleinstehenden Objekt eine Segmentnummer zugeordnet, welche in der Eigenschaft m_segment gespeichert wird. Anhand dieser Informationen können einzelne Objekte schnell erkannt und analysiert werden. Der gespeicherte Wert dient hierbei ausschließlich der Performanzverbesserung der höheren Funktionen der Gitterkarte.

m_randpixel Zusätzlich zu der Information der Zugehörigkeit zu einem Objekt sind für verschiedene Aufgaben Informationen über die Position eines Feldes innerhalb eines Objektes interessant. Die benötigte Information ist hierbei, ob sich das aktuell betrachtete Feld innerhalb eines Objektes befindet, oder am Rand eines solchen. Dies wird in der Eigenschaft m_randpixel in Form eines Boolean gespeichert. Hierbei bedeutet 'true', dass es sich um ein Feld am Rande eines Objektes handelt und 'false', dass es von allen Seiten von Feldern des gleichen Objektes umschlossen wird.

m_sondereigenschaft Bei dieser Eigenschaft handelt es sich um einen Integer, der temporär Informationen speichern kann. Diese Informationen dürfen nicht wichtig sein, da es jeder Funktion frei steht diese zu ändern. Daher kommt sie auch meist nur in Testumgebungen zum Einsatz, um bei der Fehlersuche zu helfen. Aber auch zur temporären Unterstützung von höheren Funktionen wird diese Eigenschaft verwendet. Ein Beispiel ist hierbei die Routenplanung aus Abschnitt 4.2.1. Während dieses Vorgangs wird innerhalb der Variable m_sondereigenschaft eine Schätzung der Entfernung vom Startpunkt der Route gespeichert. Im folgenden werden diese Daten ausgewertet und sind dann nicht mehr von Interesse. Ein anderes Einsatzgebiet ist die grafische Ausgabe der Karte. Hierbei können zusätzliche Informationen im Vorfeld der Ausgabe in diese Variable geschrieben werden, welche dann bei der Ausgabe als Bitmap-Datei berücksichtigt werden.

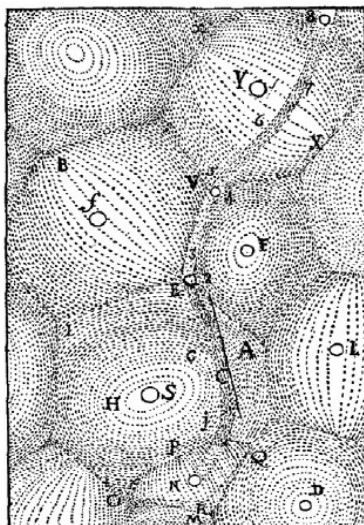


Abbildung 4.8: Descartes' Zerlegung des Raumes [14]

4.3 Voronoi

In diesem Kapitel wird über die wichtige Rolle des Voronoi-Diagramms in der Geometrie gesprochen, die in der Informatik Anwendung findet.

Zuerst werden auf historische Überlegungen eingegangen, dann das Voronoi-Diagramm definiert und zum Schluss wird ein Algorithmus zur Berechnung des Voronoi-Diagramms vorgestellt.

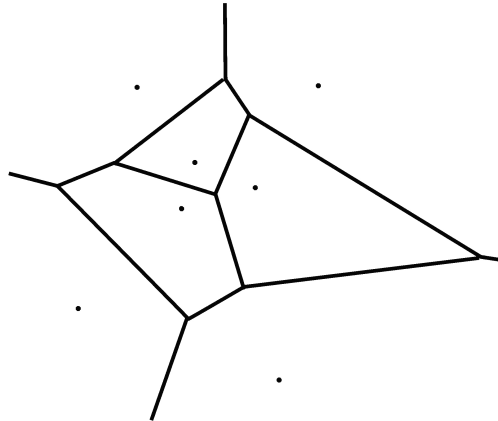
4.3.1 Einführung in das Voronoi-Diagramm

Descartes erwähnte 1644 die Vermutung, das Sonnensystem bestehe aus Materie, die um die Fixsterne herumwirbelt. Dabei wird der Raum in konvexe Regionen zerlegt. In Abbildung 4.8 erkennt man die Fixsterne, die sich in der Mitte einer jeden Region befinden. Diese Fixsterne binden Materie, die durch die Wirbel angedeutet sind.

Mag diese Theorie auch unzutreffend sein, so enthüllt sie doch den Keim der folgenden wichtigen Vorstellung: Gegeben ist ein Raum (hier der $\mathbb{R}^3$) und in ihm eine Menge S von Objekten (zum Beispiel Fixsterne), die auf ihre Umgebung irgendeine Art von Einfluss ausüben (zum Beispiel Anziehungskraft). Dann kann die folgende Zerlegung des Raumes in Regionen betrachtet werden: Zu jedem Objekt p in S wird die Region $VR(p, S)$ derjenigen Punkte des Raumes gebildet, für die der von p ausgeübte Einfluss am stärksten ist. Die Abbildung 4.8 beschreibt die von Descartes erdachte Zerlegung des Raumes.

Der große Vorteil dieses Ansatzes liegt in der Allgemeinheit. Deshalb ist dieser Ansatz unter verschiedenen Namen in vielen Wissenschaften außerhalb der Mathematik zu finden, zum Beispiel in der Biologie, Chemie oder Physiologie.

In der Mathematik hat sich zuerst Georgi Voronoi in seiner Arbeit über quadratische

Abbildung 4.9: Voronoi-Diagramm $V(S)$ [31]

Formen mit diesem Ansatz beschäftigt. Die Objekte waren regelmäßig verteilte Punkte im Raum. Der Einfluss p , den einen Punkt x auf einen anderen Punkt y ausübt, war antiproportional zum Abstand $|xy|$. Somit bestand die Region des Punktes x aus genau den Punkten, die am nächsten zu x lagen. Diese Zerlegung in Regionen wird Voronoi-Diagramm genannt.

Im folgenden werden die strukturellen Eigenschaften des Voronoi-Diagramms und die Rolle zur Lösung von geometrischen Distanzproblemen beschrieben.

4.3.2 Definition des Voronoi-Diagramms

In diesem Abschnitt wird davon ausgegangen, dass der Raum $\mathbb{R}^2$ (zum Beispiel eine Karte) betrachtet wird. Die Objekte (zum Beispiel Hindernisse auf der Karte, wie ein Stuhlbein) sind Elemente einer n -elementigen Menge $S = \{a, b, c, \dots\}$ von Punkten in $\mathbb{R}^2$, wobei nur Mengen zugelassen werden, die mindestens zwei Elemente haben. Der Einfluss p , den Punkt $a = (a_1, a_2)$ auf einen Punkt $b = (b_1, b_2)$ ausübt, wird mit Hilfe des *euklidischen Abstandes*, es können auch andere Abstandsmaße benutzt werden,

$$|ab| = \sqrt{|a_1 - b_1|^2 + |a_2 - b_2|^2}$$

berechnet.

Zusätzlich wird noch ein Bisektor definiert. Ein Bisektor von zwei Punkten $a, b \in S$ beschreibt die Menge

$$B(a, b) = \{x \in \mathbb{R}^2; |ax| = |bx|\}$$

aller Punkte im Raum $\mathbb{R}^2$, die zu a und b den gleichen (euklidischen) Abstand haben. Somit besteht der Bisektor aus den Punkten der Mittelsenkrechten von a und b , wobei die Mittelsenkrechte eine Gerade ist, die genau aus den Punkten des Bisektors besteht. Diese

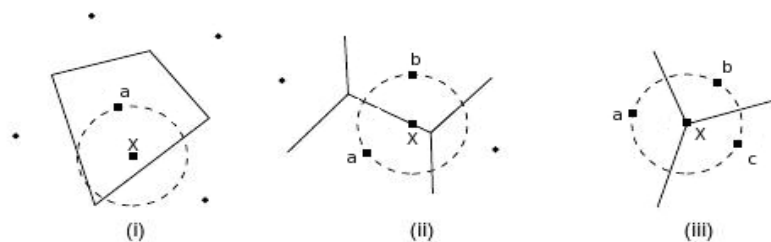


Abbildung 4.10: (i) $x \notin V(S)$ (ii) $x \in V(S)$ und ist *Kante* (iii) $x \in V(S)$ und ist *Knoten* [14]

Mittelsenkrechte teilt $\mathbb{R}^2$ in zwei offene Halbebenen. Die Halbebene, in der sich der Punkt a befindet ist

$$D(a, b) = \{x \in \mathbb{R}^2; |ax| < |bx|\}.$$

Hingegen beschreibt

$$D(b, a) = \{x \in \mathbb{R}^2; |bx| < |ax|\}$$

eine Halbebene, die den Punkt b als Element hat.

Nun kann, mit Hilfe dieser Überlegungen, eine Voronoi-Region von einem Punkt a bezüglich S definiert werden als

$$VR(a, S) = \bigcap_{x \in S \setminus \{a\}} D(a, x).$$

Da die Voronoi-Region einen Durchschnitt von $n - 1$ offenen Halbebenen beschreibt, ist sie offen und konvex.

Die Voronoi-Region $VR(a, S)$ beinhaltet alle Punkte der Ebene, die zum Punkt a näher liegen, als irgendein anderer Punkt $x \in S$.

Werden nun alle Voronoi-Regionen entfernt, bleiben am Ende nur diejenigen Punkte $x \in \mathbb{R}^2$ übrig, die zu mindestens zwei Punkten $a, b \in S$ den gleichen Abstand haben. Diese Punkte werden als Voronoi-Diagramm $V(S)$ bezüglich S bezeichnet.

Die Abbildung 4.9 beschreibt ein Voronoi-Diagramm mit sieben Punkten. Die Ebene wird in sieben konvexe Voronoi-Regionen zerlegt. Somit muss ein Punkt $x \in V(S)$ der auf dem gemeinsamen Rand von $VR(a, S)$ und $VR(b, S)$ liegt, zum Bisektor $B(a, b)$ gehören. Nun wird noch eine Klassifizierung der Punkte $x \in V(S)$ in Kantenpunkte und Knotenpunkte durchgeführt. Diese Klassifizierung wird mit Hilfe eines Kreises $K(x)$ erstellt, der den Punkt $x \in \mathbb{R}^2$ als Mittelpunkt besitzt. Dieser Kreis wird nun langsam ausgedehnt, bis der Rand des Kreises $K(x)$ auf mindestens einen Punkt $a \in S$ trifft. Dabei sind nun drei Fälle zu unterscheiden (siehe dazu Abbildung 4.10) :

1. Fall $K(x)$ trifft auf genau einen Punkt $a \in S \iff x \notin V(S)$

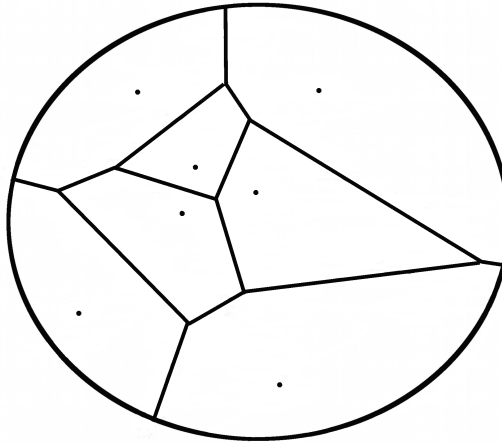


Abbildung 4.11: beschränktes Voronoi-Diagramm $V_0(S)$ [31]

- 2. Fall** $K(x)$ trifft auf genau zwei Punkte $a, b \in S \iff x \in V(S)$ und ist *Voronoi-Kante*
- 3. Fall** $K(x)$ trifft auf mindestens drei Punkte $a, b, c, .. \in S \iff x \in V(S)$ und ist *Voronoi-Knoten*

Nun lässt sich erkennen, dass ein Voronoi-Diagramm in einen ungerichteten Graphen $G = (V, E)$ übertragen werden kann. Die Menge V stellt dabei die Voronoi-Knoten und die Menge E die Voronoi-Kanten dar.

Probleme könnte noch bereiten, dass manche Voronoi-Kanten nicht beschränkt sind. Deshalb wird der wichtige Teil des Voronoi-Diagramms von einer geschlossenen “Begrenzung” H umgeben, die weit genug außen verläuft, so dass nur unbeschränkte Voronoi-Kanten mit der Begrenzung in Berührung kommen und somit beschränkt werden (siehe Abbildung 4.11). Das neu entstandene Diagramm wird als *beschränktes Voronoi-Diagramm* $V_0(S)$ definiert.

An dieser Stelle ist das Ergebnis ein kreuzungsfreier, geometrischer Graph. Das beschränkte Voronoi-Diagramm $V_0(S)$ besitzt n Voronoi-Regionen $VR(a)$ für alle $a \in S$. Jeder Knoten hat mindestens den Grad 3. Dieses gilt ebenfalls für, durch die Begrenzung H , neu entstandenen Knoten.

Eine wichtige Anwendung der Voronoi-Diagramme $V_0(S)$ ist die Routenplanung für Roboter. Es sei ein kreisrunder Roboter mit dem Radius r gegeben, der sich von dem Startpunkt s zum Zielpunkt z bewegen soll und dabei eine Berührung mit einem der n Hindernisse vermeiden muss. Die n Hindernisse werden zu der Menge S hinzugefügt und es wird eine Begrenzung H festgelegt. Falls der Roboter sich zwischen zwei Hindernisse a und b bewegen muss, ist er auf dem besten Wege, indem er zu beiden Hindernissen den *gleichen* Abstand einhält. Er sollte also dem Bisektor $B(a, b)$ folgen. Daraus lässt sich nun folgern, dass sich der Roboter innerhalb der Begrenzung am sichersten bewegen kann, wenn er den Voronoi-Kanten bezüglich $V_0(S)$ folgt. Als Vereinfachung wird angenommen, dass sich Startpunkt

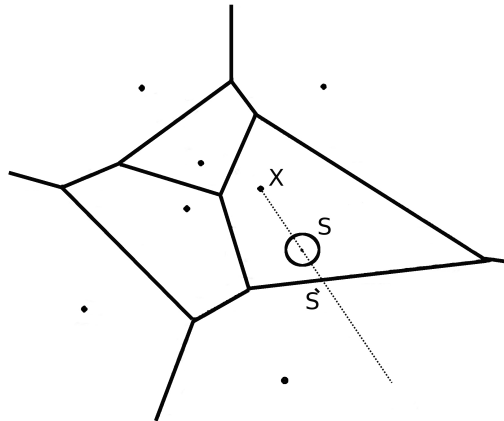


Abbildung 4.12: Der Roboter befindet sich am Startpunkt s und findet eine nahe Voronoi-Kante [31]

s und Zielpunkt z direkt auf Voronoi-Kanten befinden. Die betroffenen Voronoi-Kanten werden geteilt und s, z werden als neue Knoten hinzugefügt.

Als nächstes wird für jede Voronoi-Kante e der kleinste Abstand zu einem Hindernis berechnet. Falls die Voronoi-Kante e zum Bisektor $B(a, b)$ gehört, muss für alle Punkte $x \in e$ gelten :

$$r + \epsilon > \min(|xa|, |xb|)$$

wobei ϵ ein geeigneter Sicherheitsabstand ist. Gilt die Ungleichung nicht, ist der Weg entlang der getesteten Voronoi-Kante e zu schmal für den Roboter und wird verworfen. Nachdem alle Kanten aus den Graphen entfernt wurden, die zu schmal sind, wird vom Startknoten s eine *Breitensuche* durchgeführt. Wird nun festgestellt, dass z von s erreichbar ist, wurde ein kollisionsfreier Weg gefunden.

Im allgemeinen liegen Startpunkt s und Zielpunkt z nicht direkt auf einer Voronoi-Kante. In diesem Fall wird zuerst versucht, die Voronoi-Regionen bezüglich S zu finden, indem sich s und z befinden. Befindet sich der Startpunkt s beispielsweise in der Voronoi-Region $VR(x, S)$, wird ein Strahl ausgehend vom Hindernis x konstruiert, der durch den Startpunkt s läuft. Wie in Abbildung 4.12 zu erkennen ist, muss der Strahl irgendwann das Voronoi-Diagramm $V_0(S)$ in einem Punkt s' treffen. Nun kann sich der Roboter zum Punkt s' bewegen und es liegt die vorher besprochene Vereinfachung vor. Analog wird mit dem Zielpunkt z vorgegangen.

4.3.3 Algorithmus zur Berechnung des Voronoi-Diagramms

In diesem Abschnitt wird ein *Sweep-Line-Verfahren* eingeführt, mit dem sich ein Voronoi-Diagramm $V(S)$ in Zeit $O(n \log n)$ und Speicherplatz $O(n)$ effizient berechnen lässt.

Die Grundidee ist ganz einfach, es werden beim Sweepen (damit ist das Wandern der

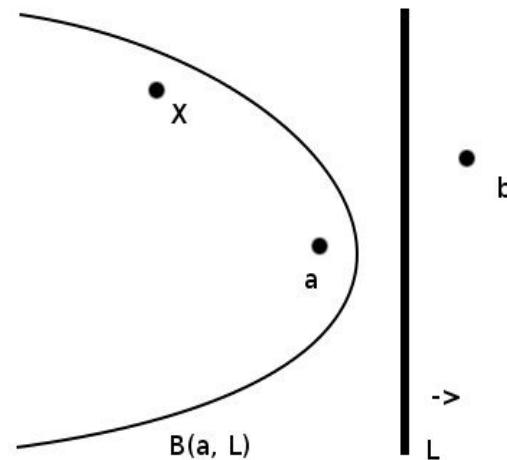


Abbildung 4.13: Kein Punkt rechts der Sweep-Line L kann das Gebiet zur Linken von $B(p, L)$ beeinflussen.

Sweep-Line in der Ebene von links nach rechts gemeint (siehe Abbildung 4.13)) nur diejenigen Teile des Voronoi-Diagramms links von der Sweep-Line gespeichert, die sich nicht mehr ändern.

Wie in Abbildung 4.13 zu sehen ist, stellt sich der Bisektor $B(a, L)$ von einem Punkt a und einer Linie L als Parabel dar. Der Punkt $x \notin S$ auf der linken Seite von $B(a, L)$ kann *nicht* zu einer Voronoi-Region rechts der Sweep-Line gehören wie zum Beispiel $VR(b)$. Diese Erkenntnis nutzt der Algorithmus.

Sei $S = \{p_1, p_2, p_3, \dots\}$ aufsteigend nach der X-Koordinate sortiert. In Abbildung 4.14 sind drei Schritte eines Durchlaufes zu erkennen. Angenommen die Sweep-Line L (die gestrichelte Linie) befindet sich im Moment zwischen den Punkten p_i und p_{i+1} . Der Rand der Voronoi-Regionen wird die *Wellenfront* W (die gepunktete Linie) genannt und besteht aus den Parabeln, die Teile von Bisektoren $B(p_j, L)$ mit $j \leq i$ sind. Links der Wellenfront sind schon Teile von Voronoi-Kanten zu erkennen (siehe Abbildung 4.14 2. und 3.), die sich im weiteren Verlauf des Algorithmus noch verlängern werden.

Während die Sweep-Line L weiter nach rechts wandert, wird die Wellenfront W für alle Parabeln $B(p_j, L)$ mit $j \leq i$ mitgezogen. Wo zwei in W benachbarte Wellenstücke sich schneiden, zum Beispiel $B(a, L)$ und $B(b, L)$, rückt der Schnittpunkt längs des *geraden* Bisektor $B(a, b)$ nach. Diese schrittweisen Verlängerungen werden Spikes genannt. Dadurch wird das Voronoi-Diagramm $V(S)$ von links in Richtung der Wellenfront W vergrößert.

Jetzt muß genauer darauf eingegangen werden, wie sich die Wellenfront W während eines Durchlaufes verändert ([4] enthält ein Applet, das einen Durchlauf zeigt):

Erstens kommt es vor, dass ein Wellenstück aus W verschwindet, weil es den Schnittpunkt seiner beiden Spikes erreicht. In Abbildung 4.14 im dritten Teilbild ist diese Situation zu erkennen. Die neu hinzugekommene Parabel wird sich in wenigen Schritten mit der Parabel des unteren Punktes verbinden, so dass der letzte Rest des Wellenstückes des linken (oder ersten) Punktes verschwindet und einen *Voronoi-Knoten* erscheint. Im weite-

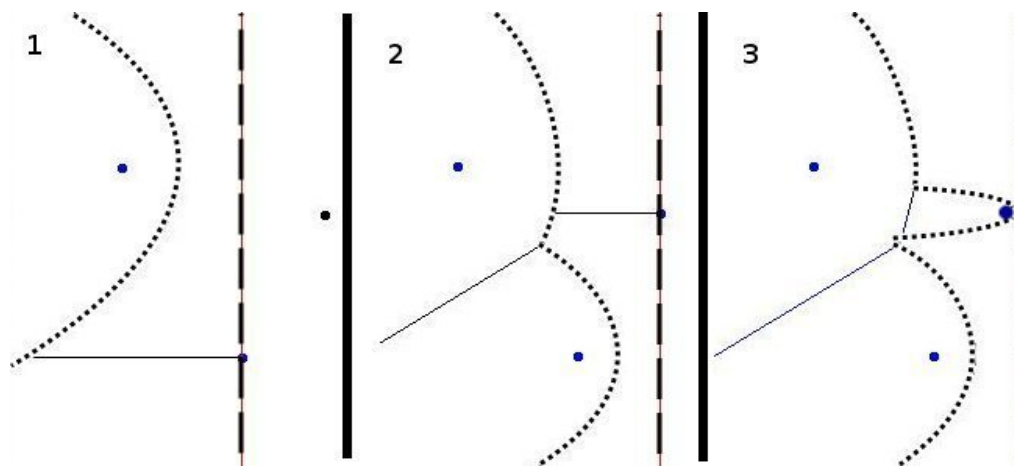


Abbildung 4.14: Drei Schritte der Berechnung eines Voronoi-Diagramms.

ren Durchlauf des Algorithmus wird die Parabel des rechten (dritten) Punktes als alleinige Wellenfront W die Voronoi-Kanten freilegen.

Zweitens können neue Wellenstücke in W erscheinen. Das passiert, wenn die Sweep-Line L auf einen neuen Punkt p trifft. Beim ersten Aufeinandertreffen ist dieser Punkt p ein Teil der Sweep-Line L und somit ist der Bisektor $B(p, L)$ eine senkrechte Linie auf der Sweep-Line L durch den Punkt p (siehe 1,2). Bewegt sich die Sweep-Line L ein Stück weiter nach rechts, erscheint eine spitze Parabel (siehe 3). Trifft diese Parabel nun auf die Wellenfront W , werden die Wellenstücke, die sich im inneren der Parabel befinden, entfernt und die beiden Schnittpunkte werden zu einer *Voronoi-Kante* verbunden (siehe siehe Abbildung 4.14 3). Der Schnittpunkt der neuen Parabel mit der Wellenfront W kann ein Punkt sein, wo sich vorher schon zwei Wellenstücke getroffen haben. In diesem Fall wird zusätzlich ein Voronoi-Knoten aufgedeckt. Zusammenfassend ist zu sehen, dass zwei wichtige Beobachtung bezüglich der Wellenfront gemacht worden sind:

- Spike-Beobachtung: Ein Wellenstück trifft auf den Schnittpunkt seiner beiden Spikes und verschwindet.
- Punkt-Beobachtung: Die Sweep-Line trifft auf einen neuen Punkt und ein neues Stück Wellenfront entsteht.

Solange keiner dieser Beobachtungen eintrifft, läuft die Wellenfront W ungestört weiter und am Ende ist ein fertiges Voronoi-Diagramm $V(S)$ zu sehen.

Zur Implementierung wurde zum einen auf den Voronoi-Diagramm Algorithmus der “MapMangerLibrary” von Shane O’Sullivan [19] und auf die “Boost Graph Library (BGL)” [5] zurückgegriffen. Die Tests des Voronoi-Diagramm Algorithmus mit einer kleinen Testmenge haben gezeigt, dass die Ergebnisse relativ ungenau sind. Mit der Hoffnung die Ungenauigkeiten später zu runden, wurden die Ergebnisse mit Hilfe Boost Graph Library weiter

verarbeitet. Die BGL bietet viele Funktionen wie zum Beispiel Dijkstra oder Breitensuche in optimaler Rechenzeit an.

Als dieser Routenplanungsalgorithmus in die Umgebung der Roboter eingefügt wurde, stellte sich heraus, dass die Ergebnisse so ungenau waren, dass eine Rundung nicht mehr möglich war und die Implementierung somit unbrauchbar ist.

4.4 Player-Stage

Player-Stage besteht aus zwei Komponenten, welche Player und Stage heißen. Player ist eine Softwareschicht die den Zugriff auf die Roboterhardware unterstützt und den Zugriff auf Netzwerkschnittstellen ermöglicht. Stage ist ein Simulator für Player, der neben dem Roboter auch diverse Sensoren zur Verfügung stellt. Mit Stage ist es möglich Software zu testen, bevor sie am realen Roboter eingesetzt wird. Neben Stage gibt es noch Gazebo einen 3D Simulator. Abbildung 4.15 gibt die Zusammenhänge wieder. Player fungiert als Server. Selbst geschriebene Programme können sich über eine Socketverbindung mit Player verbinden. Wenn eine Simulationsumgebung gestartet wurde, kann der Client das selbstgeschriebene Programm mit einem Roboter in der Simulationsumgebung mit Hilfe von Player kommunizieren lassen. Es ist auch möglich Player mit einem realen Roboter zu verbinden. Zudem ist es möglich Player über eine Konfigurationsdatei zu konfigurieren. Die Simulationsumgebungen lassen sich ebenfalls konfigurieren. Stage wird benutzt um die Steuerungssoftware für den RAPTOR zu testen. Dies ist besonders zu Beginn der Entwicklung eine Erleichterung, da der RAPTOR noch nicht fertiggestellt ist.

4.4.1 Installation

Im Folgenden wird die Installation von Player/Stage dokumentiert.

Playerinstallation

Diese Installationsanleitung beschreibt die User-spezifische Installation von Player Stage in einem Unterordner "Player" des eigenen Home-Verzeichnisses. Standardmäßig sind die Installationsroutinen von Player-Stage darauf eingerichtet, eine lokale Installation für alle Nutzer vorzunehmen. Daher mussten einige Änderungen an dem empfohlenen Installationsablauf vorgenommen werden.

1. Download der aktuellen Version (player-<version>.tar.bz2) von SourceForge nach /home/<name>/Player.
2. Wechseln nach /home/ <name>/Player.

```
$ cd /home/<name>/Player}
```

3. Entpacken:

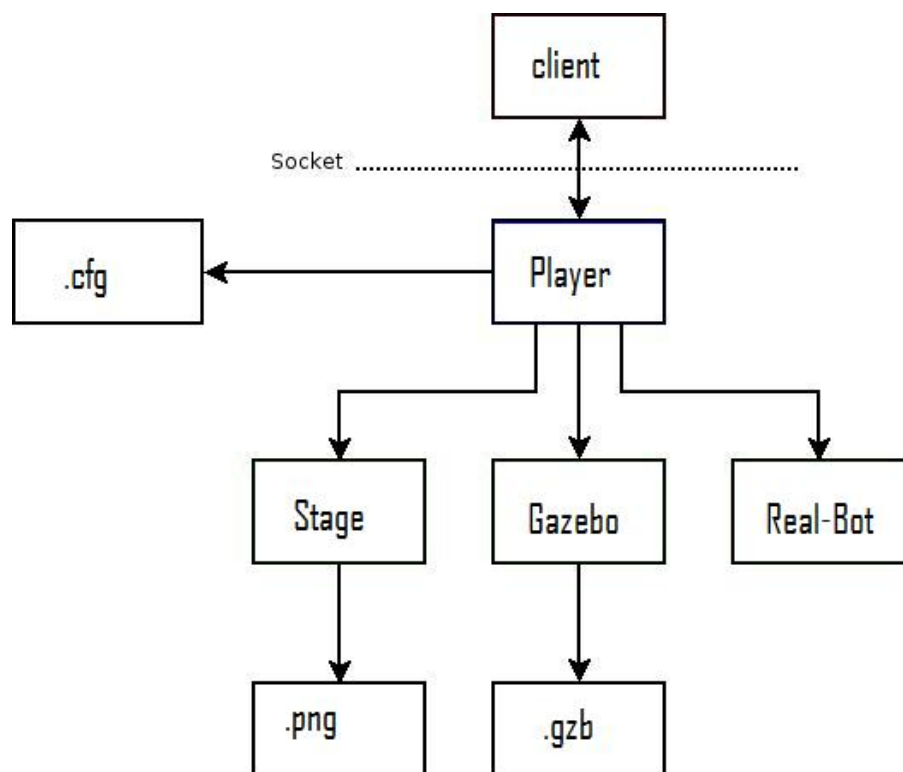


Abbildung 4.15: Komponentenuübersicht Player/Stage/Gazebo [21]

```
$ bzip2 -d player-<version>.tar.bz2
$ tar -xvf player-<version>.tar
```

4. Wechseln ins Player-Verzeichnis.

```
$ cd player-<version>
```

5. Setzen von Umgebungsvariablen.

```
$ export PATH=~ /Player/bin:$PATH
$ export CPATH=~ /Player/include:$CPATH
$ export LIBRARY_PATH=~ /Player/lib:$LIBRARY_PATH
$ export PKG_CONFIG_PATH=~ /Player/lib/pkgconfig:
    $PKG_CONFIG_PATH
$ export PYTHONPATH=~ /Player/lib/python2.2/site-packages:
    $PYTHONPATH
```

6. Überprüfen, dass Flags nicht gesetzt sind:

```
$ env | grep CFLAGS
$ env | grep LDFLAGS
```

7. Falls vorheriger Punkt eine Ausgabe verursacht, müssen die Flags deaktiviert werden:

```
$ unset LDFLAGS
$ unset CFLAGS
```

8. Konfiguration starten.

```
$ ./configure --prefix=/home/<Name>/Player
```

9. Kompilieren.

```
$ make
```

10. Installieren.

```
$ make install
```

Stageinstallation

Bevor Stage installiert werden kann, muss Player installiert und funktionstüchtig sein.

1. Download der neuesten Version von Stage vor dem Entpacken

```
$ tar xzvf stage-<version>.tgz
```

2. In das entsprechende Unterverzeichnis wechseln

```
$ cd stage-<version>
```

3. Konfiguration starten

```
$ ./configure --prefix=/home/<Name>/Stage
```

4. Kompilieren

```
$ make
```

5. Installieren

```
$ make install
```

4.4.2 Konfiguration des Roboters

Beim Starten der Simulation wird, wenn nicht bewusst geändert, der mitgelieferte Simulationsroboter Pioneer gestartet. Dieser besitzt 16 Sonarsensoren. Da der realer Roboter nur vier besitzt, jeweils einen vorne, links, hinten und rechts, muss der Pioneer an unseren realen Roboter angepasst werden. In dem Ordner “worlds” der Stage-Installation befindet sich die Datei “pioneer.inc”. In dieser wird scout 16 auf scout 4 gesetzt. Die vier Sonarsensoren werden durch Anpassen der Koordinaten an die richtigen Stellen gesetzt. Bei dem Pioneer wurden die Koordinaten dazu wie folgt gesetzt:

```
scout 4
```

```
spose[0] [ 0.170 0.0 0 ]
spose[1] [ -0.050 -0.130 -90 ]
spose[2] [ -0.250 0.0 -180 ]
spose[3] [ -0.050 0.130 90 ]
```

Durch diese Einstellungen wird ein Roboter wie aus Abbildung 4.16 generiert. Links im Konsolenfenster der Abbildung 4.16 sind die Sonardaten aus der aktuellen Position des Roboters zu sehen.

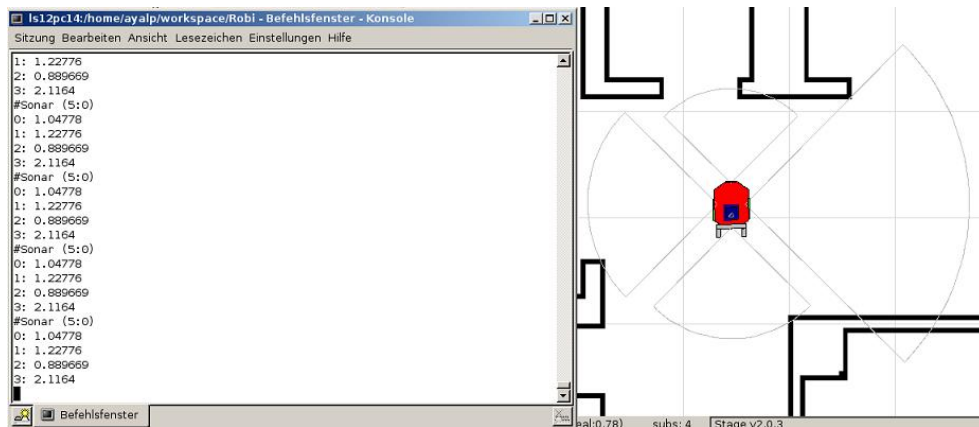


Abbildung 4.16: Simulationsumgebung

4.4.3 Simulation starten

Um Player-Stage letztendlich zu starten, müssen noch einige Umgebungsvariablen gesetzt werden. Dazu muss in der Konsole folgendes eingegeben werden:

```
export LD_LIBRARY_PATH= /home/<Name>/Player/lib
export PKG_CONFIG_PATH= /home/<Name>/Player/lib/pkgconfig/
```

Um eine einfache Welt aus Stage zu laden muss aus dem Verzeichnis

```
/home/<Name>/Player/player-<version>/server
```

die `simple.cfg` Datei wie folgt aufgerufen werden.

```
player /home/<Name>/Stage/stage-<version>/worlds/
        everything.world.cfg
```

Es stehen auch noch andere Karten zur Verfügung, wie z.B. die `simple.cfg`, die nur aus einigen Räumen und einen Roboter besteht. Diese Karte ist für unsere Zwecke ausreichend. Ist die Simulation erfolgreich gestartet worden, dann müsste eine Karte zu sehen sein wie in Abbildung 4.17. Zu sehen sind dann sechs Roboter in verschiedenen Farben, zwei Geistobjekte und eine Büroumgebung. Wände werden durch schwarze Linien dargestellt.

4.4.4 Eigene Programme erstellen

Für eigene Programme übernimmt Player die Rolle des Servers, so dass Benutzer ihre eigenen Programme zu Steuerung des Roboters schreiben können. Diese Programme bemerken keinen Unterschied zwischen einem realen Roboter und dem Roboter aus der Simulation. Die Programme, die in der Simulation den Roboter steuern, funktionieren also auch auf dem realen Roboter ohne oder mit nur geringfügigen Modifikationen.

Um die Funktionen des Simulationsroboters benutzen zu können, muss die Bibliothek

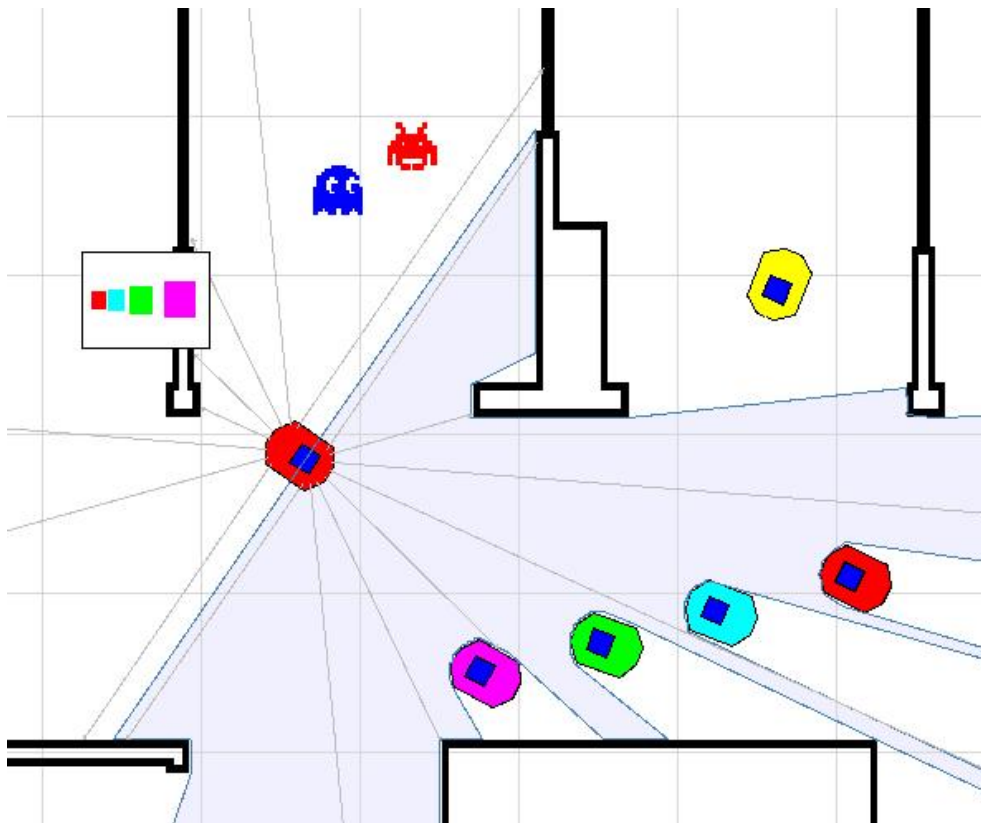


Abbildung 4.17: Simulationsumgebung

```
<libplayerc++/playerc++.h>
```

in das C++ Programm inkludiert werden. Dann stehen dem Programmierer diverse Klassen, in Player-Stage auch Proxys genannt, zur Verfügung. Zuerst wird die Verbindung zu dem Roboter mit der Klasse PlayerClient im Konstruktor der C++ Klasse hergestellt. Zudem wird der Roboter mit den Sonar, den Bumpen und den Position2dProxy verbunden. Diese sind Instanzen der Klassen SonarProxy, BumperProxy und Position2dProxy.

```
Roboter() :  
    m_roboter("localhost"),  
    m_sonarproxy(&m_roboter, 0),  
    m_position2dproxy(&m_roboter, 0),  
    m_bumperproxy(&m_roboter, 0),  
    m_mapproxy(&m_roboter, 0) {}
```

Listing 4.26: Kartengrößenausgabe

Nachdem der Roboter mit den Proxys verbunden worden ist, ist es möglich z.B. mit der Methode

```
void Roboter::setzeMotorGeschwindigkeit  
    (double x, double y, double drehrate)
```

Listing 4.27: Kartengrößenausgabe

den Roboter zu steuern.

Neben dieser Methode wurden bis jetzt auch noch folgende Methoden aus der API (siehe 4.18) implementiert:

```
// Drehrate negative Werte=rechtsKurve  
//           positive Werte=linksKurve  
//           0=geradeaus  
void setzeMotorGeschwindigkeit  
    (double x, double y, double drehrate);  
// Alle Sensordaten aktualisieren  
void leseSensorDatenEin();  
// Sensordaten sind Abrufbar  
// Beispielsweise m_sonarproxy[0]  
//   oder          m_sonarproxy[2]  
//   oder          m_position2dproxy[1] usw.  
SonarProxy* gibSonarDaten();  
// liefert true wenn irgendein Bumper aktiviert wurde  
// liefert false sonst  
bool gibKollision();  
// liefert string links, vorne oder rechts  
// bei entsprechender kollision  
std::string gibKollisionsseite();
```

```
// startet Motor
//   true=starten
//   false=ausschalten
void starteMotor(bool starten);
void setzeOdometry(double x,double y,double ausrichtung);
// holeKarte und im proxy speichern
void holeKarte();
//gibt Kartenaufloesung m/cell zurueck
double gibXGeschwindigkeit();
double gibYGeschwindigkeit();
double gibXPosition();
double gibYPosition();
double gibAusrichtung();
```

Listing 4.28: Kartengrößenausgabe

4.4.5 Makefile

Das Programm muss schliesslich kompiliert werden. Dazu wurde das folgende Makefile geschrieben. Aus den Roboter.cpp und Main.cpp wird die Main.bin erstellt.

```
all : Roboter.o Main.o
    g++ Main.o Roboter.o -o Main.bin `pkg-config \
        --cflags playerc++` `pkg-config --libs playerc++`

Roboter.o : Roboter.cpp
    g++ `pkg-config --cflags playerc++` -c Roboter.cpp

Main.o: Main.cpp
    g++ `pkg-config --cflags playerc++` -c Main.cpp

clean:
    rm -f *.o Main.bin
```

Mit dem Aufruf Main.bin startet die Simulation des programmierten Roboters, vorausgesetzt die Simulationsumgebung Stage wurde, wie bereits beschrieben, gestartet.

4.5 Application Programming Interface

In Abbildung 4.18 ist der erste Entwurf unserer API zu sehen. Diese gibt die Struktur unserer Steuerungssoftware wieder. In größeren Gruppen, wie in unserer PG, ist es wichtig in einer API die Softwarestruktur im Modell festzuhalten. Dadurch hat jedes Mitglied den

Überblick über die zur Zeit bereits geschriebenen und noch nicht fertiggestellten Quellcodes. Die API wird in Player-Stage nur teilweise implementiert, da z.B. die Lichtquelle oder der Staubsauger zu Simulationszwecken nicht benötigt werden. Dem realen Roboter wird diese natürlicherweise vollständig zur Verfügung gestellt. Die API besteht zur Zeit aus zwei Komponenten, den Sensoren und den Aktuatoren. Die Sensoren erben von der Klasse Sensor und die Aktuatoren von der Klasse Aktor. Zur Zeit steht noch nicht fest welche gemeinsamen Attribute und Methoden die Sensoren oder die Aktuatoren erben, daher sind in Abbildung 4.18 in den entsprechenden abstrakten Vaterklassen noch keine Attribute oder Methoden festgehalten. Da die erbbenden Klassen von einer Vaterklasse erben, ist es in der späteren Programmierung wie z.B. bei der Sensordatenfunktion möglich, die Sensoren in ein Array zu speichern, um so komfortablen Zugriff auf einzelne Sensoren zu erhalten. Die einzelnen erbbenden Klassen sind so konzipiert, dass grundlegende Informationen für eine Sensordatenfusion zu Verfügung stehen. Die Klasse Ultraschallsensor stellt Methoden zur Verfügung die die gemessene Entfernung eines Sensors und dessen Position relativ zur Ausrichtung des Roboters wiedergeben. Die Klasse Kompass stellt eine Methode zu Verfügung die die eigentliche Ausrichtung des Roboters wiedergibt. In einem Objekt der Klasse Bumper wird, wenn der dazugehörige echte Bumper aktiviert wird, die Zeit der Aktivierung gespeichert. Andere Programmmodule wie z.B. die Robotersteuerung können über die Klasse Antrieb direkt auf den Motor zugreifen und somit den Roboter zum fahren oder anhalten über die entsprechenden Methoden bringen. Die Rückgabewerte der einzelnen Methoden und die Datentypen der Attribute wurden noch nicht festgelegt. Dies wird im Laufe der Entwicklung geschehen.

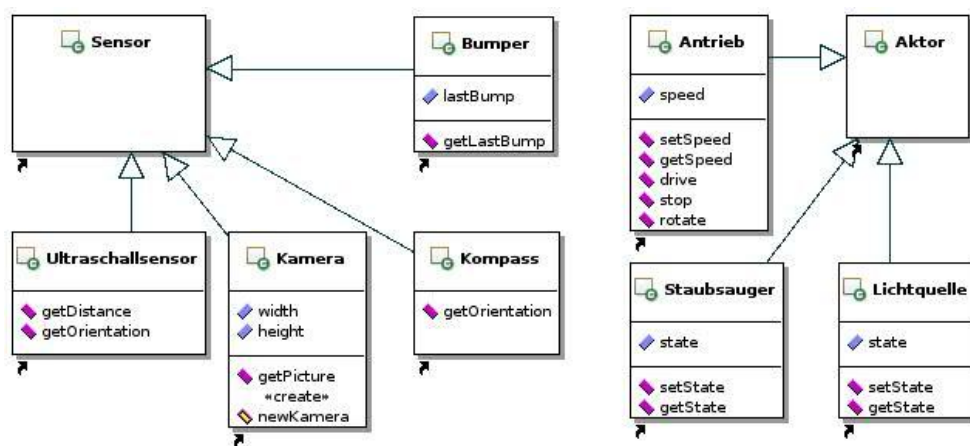


Abbildung 4.18: API

4.6 ATmega32

Im Folgenden wird nur die Software des ATmega32 für den Maussensor und die Motoren beschrieben. Die restliche Software ist Teil des zweiten Semesters der Projektgruppe.

4.6.1 Maussensor

Auf der Webseite von Roboternetz.de befand sich bereits ein Beispielprogramm ([23]) zur Ansteuerung des im Raptor verwendeten Maussensors. Dieses konnte relativ schnell angepasst werden.

Die Platine hat bereits die Kontakte SDA und SCL - was sehr analog zum I^2C -Bus aussieht. Tatsächlich ist das Protokoll bis auf einige Kleinigkeiten sehr ähnlich:

- Es ist kein Bus, sondern eine Direktverbindung
- Somit ist keine Hardware-Adressierung nötig und
- auf die Registeranwahl vom Controller aus folgt sofort die Antwort des Sensors - ohne dass ein Restart nötig wäre.
- Das Bit für die Auswahl der Datenrichtung ist invertiert.
- Die 5V-Default-Spannung auf dem Bus wird vom Controller vorgegeben.

Damit lässt sich die Platine nicht direkt an den I^2C -Bus anschließen. Auf der anderen Seite bedeutet das, dass zwei beliebige Pins verwendet werden können. Für das Beispiel wurde SCK mit Pin 40 (PortA.0) und SDIO/SDA mit Pin 39 (PortA.1) des Mikrocontrollers verbunden.

Der abgewandelte Quelltext befindet sich im Anhang unter Listing B.17

Nach der Initialisierung kann nun aus Register 2 und 3 die Y- bzw. X-Verschiebung seit dem letzten Auslesen ausgelesen werden. Um daraus eine aktuelle Position zu ermitteln, müssen die Werte zusammengerechnet werden. Da die empfangenen Daten *signed* sind, reicht eine einfache Addition.

Aus dem Maussensor kann auch komplett das zuletzt aufgenommene Bild ausgelesen werden. (Register 08 mit 1 beschreiben, dann $324 = 18 \times 18$ mal Register 08 auslesen) Es sind nur die unteren 6 Bit mit Bilddaten gefüllt sind, dies entspricht einem Graustufen-Bild mit 64 Graustufen. Bisher konnte der ATmega32 dieses Bild nicht ausgeben oder weitergeben und es ist stark zu bezweifeln, dass diese Funktionalität im Roboter sinnvoll eingesetzt werden kann. Dennoch ist sie vorhanden und kann einfach implementiert werden.

4.6.2 Motoren

Das Fahrwerk des Roboters besteht aus 2 Motoren des Modells RB 35, welche dem Roboter nach geeigneter Bahnplanungsberechnung ermöglichen sollen, die gesamte Fläche eines Raumes zu befahren. Hierzu soll die Geschwindigkeit und die Drehrichtung der Motoren und insbesondere die Fahrtrichtung des Roboters steuerbar sein, was durch einen entsprechenden Motortreiber realisiert wird.

Der Mikrocontroller selber besitzt keine Möglichkeit zur direkten Spannungsregelung und die Einschränkung auf eine einzig mögliche konstante Geschwindigkeit würde zu unnötigen Einschränkungen führen und nicht den Vorstellungen eines effektiven Saugroboters

entsprechen. Außerdem würde die Umsetzung durch ein entsprechendes Bauteil, welche die Spannung regelt, den Aufwand und die Kosten unnötig erhöhen. Aus diesen Gründen fiel die Entscheidung auf den Motortreiber L293D (Kapitel 3.5), welcher schon bei vielen Roboter-Projekten verwendet wurde, da er viele Vorteile bietet.

Mit diesem Motortreiber ist es möglich zwei Motoren gleichzeitig zu steuern. Die Eingänge sind TTL-Kompatibel (Transistor-Transistor-Logik) und der Baustein hat einen integrierten Überspannungs- und Temperaturschutz. Der Spannungsbereich liegt zwischen 4,5V und 36V und der Ausgangsstrom pro Ausgang beträgt 500mA. Die zwei Kanäle bestehen aus je 3 Eingängen und 2 Ausgängen, womit 2 Motoren gesteuert werden können. Die zwei A-Eingänge pro Kanal bestimmen durch unterschiedliche Polarität die Drehrichtung des Motors, bei zwei gleichen Signalen (LOW oder HIGH) stoppt der Motor. Zudem gibt es pro Kanal noch einen EN-Eingang, wodurch erst bei gesetztem HIGH-Signal ein Stromfluss entsteht, der den Motor vorantreibt. Zur Spannungsversorgung besitzt der Baustein 2 Eingänge VCC1 und VCC2. VCC1 liefert den Strom für den Baustein selber, während durch den VCC2 -Eingang eine externe Spannungsversorgung der Motoren realisiert werden kann. Diese Funktionalität wird ausgenutzt, um den Motor mit den durch den gewählten Akku maximal möglichen 12V (statt 5V, welche für die restlichen Schaltungen des Roboters verwendet werden) zu betreiben.

Motorsteuerung durch den Mikrocontroller mittels Pulsweiten-Modulation

Die Eingänge 1A und 2A für den linken Motor und 3A und 4A für den rechten Motor werden vom ATmega32 digital beschaltet, um die Drehrichtung festzulegen. Die Enable-Eingänge 1,2EN für den linken Motor und 3,4EN für den rechten Motor müssen somit ebenfalls beschaltet werden und die maximale Fahrtgeschwindigkeit des Roboters wird durch die an VCC2 anliegende Spannung gesteuert. Wünschenswert ist hierbei noch eine Regulierung der Fahrtgeschwindigkeit.

Die häufigste Methode ist die Realisierung mit einer Pulsweiten-Modulation (PWM). Bei der Pulsweiten-Modulation werden Impulse mit voller Spannung aber variabler Breite an einen Verbraucher gesendet. Dieses modulierte Rechtecksignal wird in einer konstanten Frequenz generiert. Die Breite der Schaltimpulse wird durch das Tastverhältnis (das Verhältnis der Länge eines HIGH- zu einem LOW-Signal innerhalb einer Periode) bestimmt. Dies hat den Vorteil, dass keine Spannung geregelt werden muss. Das PWM-Signal wird den beiden EN-Eingängen des Motortreibers angelegt und durch die Trägheit der Motoren kann, bei entsprechender Pulsbreite, die Geschwindigkeit reguliert werden. Der ATmega32 unterstützt das Erzeugen von PWM-Signalen direkt mittels seiner 3 internen Zähler. Timer1 ist hierbei am interessantesten für die Motorsteuerung des Roboters, da mittels dieses einen Zählers, zwei unterschiedliche PWM-Signale generiert werden können. Die Pins OC1A (Pin D5) und OC1B (Pin D4) des ATmega32 dienen hierbei als Ausgänge der PWM-Signale, welche jeweils mit den Enable-Eingängen 1,2EN und 3,4EN verbunden sind.

Die PWM-Einstellung des ATmega32

Der interne Zähler Timer/Counter1 des ATmega32 kann genutzt werden, um ein PWM-Signal zu erzeugen. Je nach Einstellung beginnt der Timer bei 0 zu zählen und setzt gleichzeitig einen Ausgangspin auf HIGH. Sobald der Zähler einen (vorher festgelegten) Wert erreicht, wird der Timer zurück und dabei der Ausgangspin auf LOW gesetzt. Dieser Wert ist bestimmt durch die Output Compare Unit. Der Zählerwert im Register TCNT1 wird mit dem Wert im jeweiligen Output Compare Register OCR1A oder OCR1B verglichen und bei einem Match dementsprechend gesetzt. Dieser Vorgang wiederholt sich immer wieder und durch das Verändern des Vergleichswertes im Register OCR1A oder OCR1B, wird Einfluss auf das Verhältnis von HIGH- und LOW- Phasen der Ausgangspins während einer Zählperiode genommen.

Der Vorteil des Timer/Counter1 ist somit, dass er 2 unterschiedliche Ausgänge beschalten kann, so dass beide Motoren mit demselben Zähler reguliert werden können. Der Mikrocontroller stellt mehrere Modi für die Generierung einer Pulsweiten-Modulation zur Verfügung. Eine entsprechende PWM wird erzeugt, in dem der Waveform Generation Mode angepasst und die Einstellung des Compare Output Mode bestimmt wird. Diese Einstellungen bestimmen wie der Ausgang beschaltet werden soll, wenn es ein Matching zwischen TCNT1 und OCR1x gibt. Die Bits WGM10 bis WGM13 bestimmen die Einstellung des Waveform Generation Mode. Wenn diese auf Normal Mode gesetzt sind zählt der Zähler bis zu seinem Maximum von 16 Bit (MAX = 0xFFFF) und startet danach wieder bei 0 (BOTTOM = 0x0000) während ein Overflow-Flag gesetzt wird. Dieser Modus ist aber zur Motorsteuerung ungeeignet, da das Ereignis mit einem Interrupt abgefangen werden müsste, wodurch eine Routine gestartet werden würde, die dann den Ausgang umschaltet. Außerdem müsste der Zählerstand manipuliert werden, um die Pulslänge zu verändern. Da aber der Modus zu viel CPU-Zeit beansprucht, wird sogar vom Hersteller davon abgeraten, ihn zu nutzen. Des Weiteren gibt es noch einen Clear Timer on Compare Match Modus. Hierbei bestimmen die Register OCR1A oder ICR1 den maximalen Zählerwert, bei dessen Erreichen der Ausgang invertiert wird. Somit ist es möglich, direkten Einfluss auf die Frequenz der PWM zu nehmen und der eingestellte Wert entspricht dann dem Pulsweitenverhältnis. Der dritte Modus ist der Phase Correct PWM Mode. Der Zähler zählt hierbei von BOTTOM bis TOP und wieder von TOP zu BOTTOM zurück. Im nicht invertierenden Compare Output Mode wird bei einem Matching zwischen TCNT1 und OCR1x beim Hochzählen der Ausgang gelöscht und beim Herunterzählen gesetzt, im invertierenden Modus dementsprechend entgegengesetzt. Außerdem gibt es noch einen Phase and Frequency Correct PWM Mode, welcher sich kaum von dem vorherigen Modus unterscheidet. Viel interessanter für das RAPTOR-Projekt ist der Fast PWM Mode. Bei dieser Einstellung zählt der Zähler von BOTTOM bis TOP und startet dann wieder bei BOTTOM. Von daher kann die Frequenz im Fast PWM Mode doppelt so hoch sein im Vergleich zum Phase (and Frequency) Correct Mode, was den Anforderungen einer Motorsteuerung eher entspricht. Der Aufwand ist somit geringer und es wird praktisch keine CPU-Zeit beansprucht, wie z.B. im Normal Mode. Der Fast PWM Mode wird mit 8 Bit, 9 Bit oder 10 Bit betrieben, so dass der Zähler entsprechend von 0 bis 255, 511 oder 1023 hochzählt. Im nicht-invertierenden

Output Compare Mode wird bei einem Matching zwischen TCNT1 und OCR1x der Ausgang auf LOW gesetzt und, sobald der Zähler wieder bei BOTTOM beginnt zu zählen, auf HIGH gesetzt. Die Pins OC1A und OC1B müssen aber auf Ausgang gerichtet werden, damit das PWM-Signal verwendet werden kann. Die Frequenz des PWM-Signals wird folgendermaßen berechnet:

$$f_{OCn \times PWM} = \frac{f_{clk}}{N \cdot (1 + TOP)}$$

f_{clk} bezeichnet die Taktfrequenz des Mikrocontrollers, welche bei 16Mhz liegt und N ist der einstellbare Wert des Frequenzteilers, welcher auf 1024 gesetzt wurde. Der Frequenzteiler (Prescaler) teilt die Taktfrequenz in einem bestimmten Verhältnis auf.

Mikrocontroller Code

Die Drehrichtung wird durch den Treiber bestimmt. Die Pegel, die an die Pins des L293D gelegt werden, bestimmen hierbei die Richtung wenn 1Y und 2Y die Eingänge für den linken und 3Y und 4Y für den rechten Motor sind. Dies gilt nur, wenn 1A bzw. 3A am Pluspol des jeweiligen Motors angeschlossen ist:

1Y	2Y	Aktion
0	0	bremsen
1	0	vorwärts
0	1	rückwärts
1	1	bremsen

Analog dazu gilt das Gleiche auch für den rechten Motor, wobei dieser umgekehrt geschaltet wird.

Der vollständige Quelltext zur Ansteuerung des Motortreibers befindet sich als Listing B.18 im Anhang.

Bemerkungen:

Zeilen 5-16 Statt den Präprozessoranweisungen sind softwareseitig auch direkte Variablenzuweisungen anhand von Pins und Ports möglich, doch diese zuvor zu definieren ermöglicht einen späteren Wechsel und hält den Quelltext wesentlich übersichtlicher.

motor_init initialisiert die Motorsteuerung des ATmega32.

MOTOR_DIR_DDR setzt die Datenrichtungsregister der als PWM-Ports geschalteten Pins. Die Datenrichtungsregister für PC7 und PC6 (linker Motor) und PC5 und PC4 (rechter Motor) werden auf HIGH gesetzt, womit festgelegt wird, dass diese Pins Output erzeugen. Diese Pins steuern hinterher die Richtung des jeweiligen Motors durch den Treiber L293D

TCCR1A und TCCR1B richten den Timer ein. Der Waveform Generation Mode wird dabei auf "8 Bit Fast PWM Mode" gesetzt. Der Compare Output Mode

wird auf “non-inverted Compare Output Mode” gesetzt. Dies bedeutet, dass bei einem “Compare Match” der jeweilige Ausgang auf LOW gesetzt wird und bei Wiederbeginn des Zählers auf HIGH. Dadurch lässt sich die PWM-Frequenz durch die Werte in den Output Compare Registern OCR1A und OCR1B steuern. Der Wert darf wegen dem 8 Bit Zähler zwischen 0 und 255 liegen, wobei 0 Stillstand bedeutet und 255 die maximale Geschwindigkeit. Desweiteren wird der Prescaler (Frequenzverteiler) auf 1024 gesetzt.

TCNT1 bis Ende der Initialisierung: Der Timer1 wird mit 0 vorgeladen, um Fehler bei der Initialisierung zu vermeiden. Die Datenrichtungsregister der PWM-Pins OC1A und OC1B müssen auf Output eingerichtet werden. Hierüber werden die PWM-Signale ausgegeben

motor_drive muss aufgerufen werden, damit der Roboter sich vorwärts bewegt. Hierbei müssen geeignete Parameter übergeben werden. Als Eingabe sind pro Motor nur Werte zwischen 0 (Stillstand) und 255 (maximale Geschwindigkeit) zulässig. Somit bremst der Roboter beim Wert 0. Gegebenfalls wird noch eine Methode implementiert, die einen Kurzschluss der Motoren verursacht, um eine starke Bremse zu erzwingen. Doch das Eigengewicht des Roboters und die daraus entstehende Reibung der Räder mit dem Boden führen bereits beim Umschalten auf eine geringe PWM-Frequenz zum Stillstand. Desweiteren kann die Fahrtrichtung mit einem booleschen Wert festgelegt werden. Bei true fährt der Roboter geradeaus, bei false rückwärts. Somit ist es möglich mit konstanter Geschwindigkeit geradeaus oder rückwärts zu fahren. Außerdem lassen sich beide Motoren unabhängig voneinander mit unterschiedlichen Geschwindigkeiten ansteuern, so dass es auch möglich ist Kurven zu fahren.

motor_turn_left und motor_turn_right veranlassen den Roboter zu einer Linksdrehung bzw. Rechtsdrehung. Dabei drehen sich die Räder in entgegengesetzte Richtungen. In Zukunft wird der Roboter sogar in der Lage sein, sich um einen bestimmten Winkel exakt zu drehen. Die Implementierung folgt, sobald wir am fahrenden Roboter testen können.

4.6.3 Inter-Integrated Circuit

Der Inter-Integrated Circuit (I^2C , auch IIC oder Inter-IC)-Bus ist ein serieller Zweidraht-Bus, der auf Grund seiner Vorteile heutzutage in vielen Systemen verwendet wird. Als er vor etwa zwanzig Jahren von der Firma Phillips entwickelt wurde, waren die Ziele, mit möglichst wenig Adern auszukommen. Damit konnten Pins an den ICs, Leiterbahnen und somit auch wertvolle Fläche gespart werden. Inzwischen hat sich dieser Bus zu einem Industriestandard in unzähligen Embedded-Lösungen entwickelt.

Verwendung im Raptor

Im Raptor wird der I^2C -Bus zur Kommunikation zwischen dem Gumstix, dem ATmega32-Mikrocontroller und dem Kompassmodul CMPS03 verwendet.

- In Bezug auf den Mikrocontroller bietet I^2C den Vorteil, dass auf einen bestehenden Standard zurückgegriffen werden kann und keine neuen Methoden zur Kommunikation überlegt werden müssen. Als Alternative war vorübergehend der RS232-Standard im Blickfeld, dieser hätte aber große Nachteile im Bereich der Geschwindigkeit gehabt. Außerdem besteht so die Möglichkeit, den UART-Port des Mikrocontrollers für Debug-Ausgaben oder andere Anwendungen zu nutzen. Ein zusätzlicher Vorteil ist, dass beide Bausteine Anschlüsse für den Bus anbieten und das Protokoll direkt unterstützen.
- In Bezug auf den Kompass bietet I^2C den Vorteil, dass die Orientierung direkt ausgelesen werden kann und nicht erst aus einem PWM-Signal ermittelt werden muss. Da der Kompass direkt an den I^2C -Bus angeschlossen wird, müssen dafür keine zusätzlichen Pins am Mikrocontroller oder am Gumstix belegt werden.

Die Technik

Der Bus verwendet zwei Leitungen. Eine Leitung (SCK, Serial Clock) wird für die Takung verwendet, die andere Leitung (SDA) für die eigentliche Datenübertragung. Beide Leitungen werden über Pull-Up-Widerstände auf einem hohen Spannungspegel gehalten. In vielen Anwendungen sind dies 5V; im Raptor erzwingen die Spezifikationen des Gumstix allerdings einen Pegel von 3.3V. Weder der Mikrocontroller, noch das Kompassmodul hatten damit Probleme.

Diese beiden Leitungen sind parallel an die entsprechenden Anschlüsse der Bus-Geräte angeschlossen, somit sind also alle SDA-Pins aller Geräte miteinander verbunden. (Abbildung 4.19)

Die Kommunikation zwischen den Geräten geschieht nun dadurch, dass Geräte diese "Leitungen auf GND ziehen". Durch die implizite Verundung lesen *alle* Geräte einen LOW-Pegel, wenn auch nur *ein* Gerät die Leitung auf LOW gezogen hat. Umgekehrt lesen *alle* Geräte einen HIGH-Pegel, wenn *kein* Gerät die Leitung auf LOW zieht.

Gültigkeit von Daten

Grundsätzlich gilt, dass ein Bit auf der SDA-Leitung nur dann gültig ist, wenn es während eines gesamten SCK-Impulses konstant bleibt. Wenn der Taktgeber selbst die Daten auf den Bus legt, so wird er sicherlich selber darauf achten. Wenn der Datengeber aber nicht gleich dem Taktgeber ist, so darf er seinen Pegel nicht zwischen der steigenden Flanke und der fallenden Flanke auf der SCK-Ader ändern. (Abbildung 4.20)

Eine Ausnahme von dieser Regelung sind die Start- und die Stopp-Bedingung, die gerade dadurch zu erkennen sind, dass der SDA-Pegel während eines SCK-Impulses geändert wird. (Abbildung 4.21)

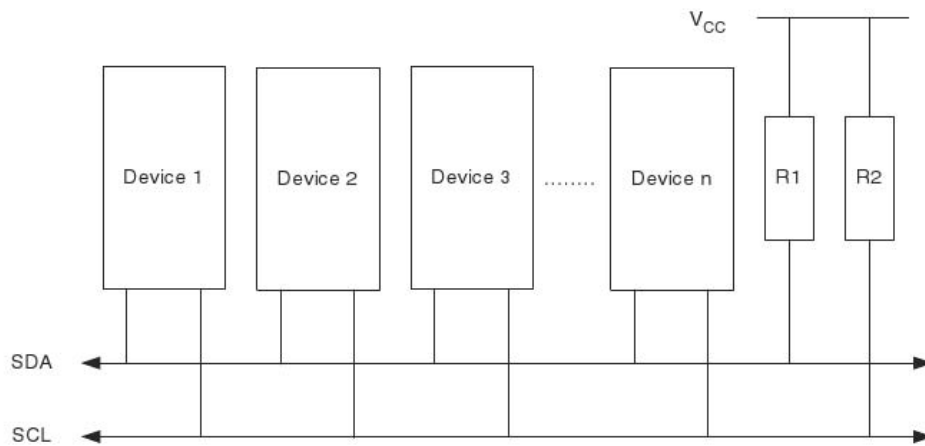


Abbildung 4.19: Anbindung von Geräten am I^2C -Bus [23]

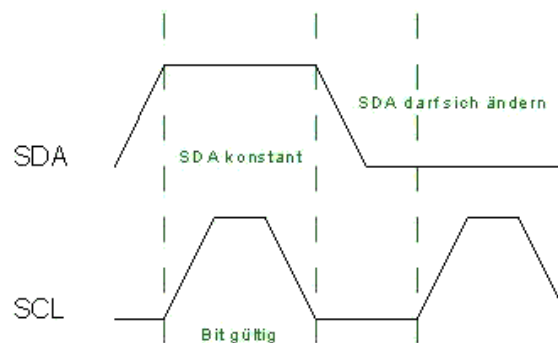


Abbildung 4.20: SDA darf sich während eines SCK-Impulses nicht ändern [23]

Startbedingung Zieht ein Gerät die SDA-Leitung auf GND, obwohl SCK auf HI liegt (zum Beispiel zwischen einzelnen Kommunikationspaketen), so signalisiert es damit, dass es zum Master auf dem Bus werden und eine Kommunikation beginnen möchte.

Stoppbedingung Ein ansteigender SDA-Pegel bei SCK-Leitung auf HI-Niveau signalisiert das Ende eines Kommunikationspaketes. Der Bus ist somit wieder frei für die nächste Kommunikation.

“Repeated Start” Wenn ein Master zwar eine Kommunikation beenden, aber sofort eine neue beginnen möchte, ohne den Bus freizugeben, so kann er statt der Stoppbedingung auch eine erneute Startbedingung senden. Dies wird dann als “Repeated Start” bezeichnet. In der Praxis ist solch ein Vorgehen sinnvoll, weil eine Einzelkommunikation nur lesend oder schreibend sein kann, jedoch keine Mischung davon. Wenn also ein Master zunächst ein Datum schreiben und dann vom gleichen Slave ein Datum lesen möchte, so ist es zu empfehlen, nicht erst den Bus freizugeben. Dieser “gemischte” Zugriff mag zunächst unsinnig erscheinen, allerdings möchte der Master dem Slave oftmals mitteilen, *welches* Datum (z.B. Register) er lesen möchte. Hier ist also ein vorheriger Schreibzugriff nötig.

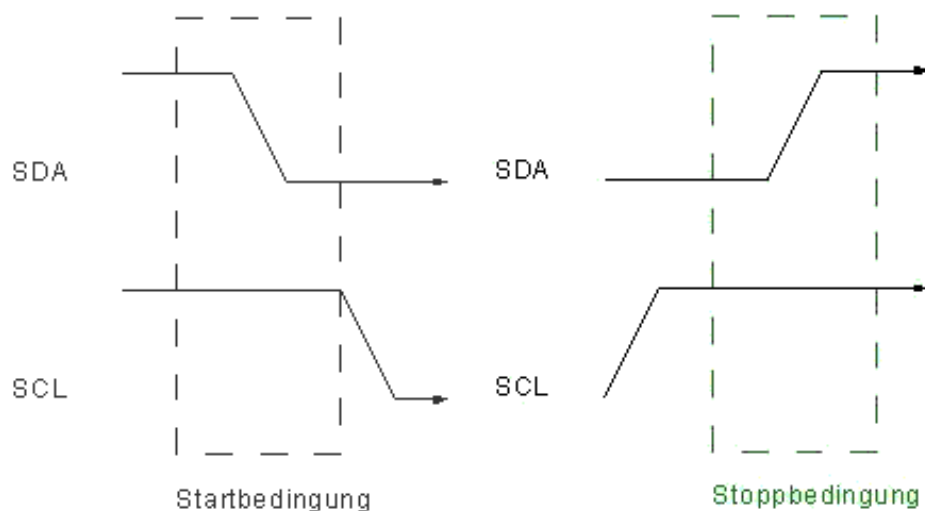


Abbildung 4.21: Start- und Stopp-Bedingung [23]

Sollten zeitgleich mehrere Geräte versuchen, eine Kommunikation zu beginnen und Master auf dem Bus zu werden, so werden die beiden (oder mehr) Geräte diese Kollision erkennen und durch eine Arbitrierung einen einzigen Master ermitteln. Im Raptor wird ausschließlich der Gumstix die Master-Rolle übernehmen, so dass es hier nicht zu Kollisionen kommen kann.

Somit wäre auch die Idee des “Repeated Starts” zu vernachlässigen, sie scheint aber dennoch eine gewisse Programmierästhetik zu beinhalten.

Bestätigung von Daten

Ein Datenpaket besteht immer aus 8 Bit. Nachdem der Sender diese 8 Bit übermittelt hat, folgt noch ein weiterer SCK-Impuls, in diesem Impuls greift der Sender nicht schreibend auf die SDA-Leitung zu, statt dessen obliegt es der Verantwortung des Empfängers, den SDA-Pegel auf GND zu ziehen - natürlich wieder, bevor auf der SCK-Leitung die ansteigende Flanke erscheint; das ACK-Bit (Acknowledge) ist in diesem Falle ein Bit wie jedes andere auch. Dieses ist dann die Bestätigung, dass der Empfänger die Daten erhalten und gespeichert hat. (Abbildung 4.22)

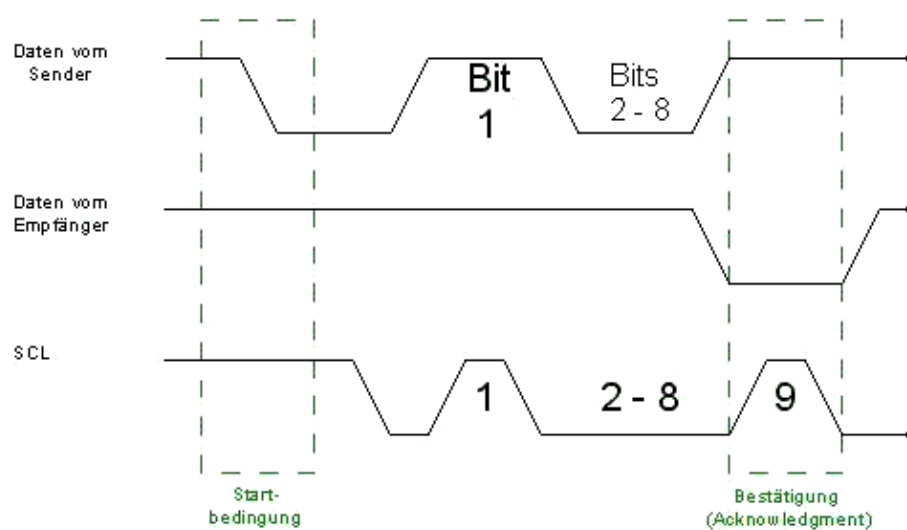


Abbildung 4.22: Der Empfänger quittiert den Empfang eines Paketes, indem er die SDA-Ader beim "9. Bit" auf Null zieht. [23]

Wenn der Empfänger den SDA-Pegel ebenfalls HIGH lässt, so ist das für den Sender ein Zeichen, dass der Empfänger die Daten nicht oder nicht richtig erhalten hat. Wie in diesem Falle zu verfahren ist, ist von der Software abhängig.

Allerdings tritt dieser NACK-Fall (Not ACK / Not Acknowledge) nicht nur bei Fehlern auf, sondern auch, wenn der Empfänger keine weiteren Daten erhalten *will*. Dieser Fall tritt beispielsweise bei dem im Raptor verwendeten Kompassmodul auf. Nachdem es adressiert und Register 2 ausgewählt wurde, sendet das Modul nun nacheinander die Register 2,3,4 usw. Da allerdings nur Register 2 und 3 relevant sind, sollte nach dem zweiten ausgelesenen Byte ein NACK gesendet werden.

Natürlich ist auch die danach gesendete Stoppbedingung ein eindeutiges Zeichen für das Ende der Kommunikation, allerdings kann das NACK sinnvoll sein, weil in manchen Anwendungen das Bereitstellen zusätzlicher Bytes für den Sender mit Zusatzaufwand verbunden ist. Durch das Senden des NACK weiß der Sender somit, dass er diesen Zusatzaufwand nicht betreiben muss, da diese zusätzlichen Bytes nicht abgefragt werden.

Adressierung

Das erste Byte nach der Startbedingung ist die Adressierung des Slaves. Der Master wählt also aus, mit welchem Slave er kommunizieren möchte, zudem wird an dieser Stelle auch schon der Modus (lesend/schreibend) definiert.

I^2C -Geräte, die als Slaves fungieren können, haben eine Adresse, auf die sie reagieren. Diese lässt sich - je nach Gerät - frei vergeben (wie im Mikrocontroller möglich), teilweise frei vergeben (manche ICs der gleichen Baugruppe sind immer im gleichen "Adressraum", nur die letzten Bits können manipuliert werden) oder kann auch fest in der Hardware vorgegeben sein (wie im Kompassmodul). Nach der ursprünglichen Definition des Bus hat eine Adresse exakt sieben Bit. Einige Bitkombinationen sind jedoch für andere Zwecke reserviert, so dass nach dieser Reservierung "nur" 114 Geräte angesteuert werden könnten. In manchen Anwendungen reichen diese 114 Adressen nicht aus. Es gibt daher neben der Möglichkeit, mehreren Slaves die gleiche Adresse zu geben, inzwischen auch eine erweiterte 10-Bit-Adressierung. Da im Raptor nur drei Teilnehmer und nur zwei Slaves am Bus betrieben werden sollten, wurde eine Einarbeitung in diesen Bereich nicht als notwendig angesehen.

Da von einem Datenbyte nur die ersten 7 Bit für die Adresse benötigt werden, wird mit dem achten Bit die Datenrichtung ausgewählt:

- Ist das Bit gesetzt, so wird lesend auf den Slave zugegriffen. Der Master befindet sich im *Master Receiver*-Modus, der Slave wird in den *Slave Transmitter*-Modus geschaltet.
- Ist das Bit nicht gesetzt, so wird schreibend auf den Slave zugegriffen. Der Master befindet sich im *Master Transmitter*-Modus, der Slave wird in den *Slave Receiver*-Modus geschaltet.

Das neunte Bit ist dann - wie schon erwähnt - das ACK-Bit. Hiermit bestätigt der Slave, dass er seine Adresse empfangen hat und "einsatzbereit" ist. I^2C sieht auch eine Möglichkeit vor, allen Slaves am Bus eine Nachricht zukommen lassen - "General Call". Hierbei senden dann alle Slaves ein ACK, durch die implizite Verundung der SDA-Ader ergibt sich dabei kein Problem.

In manchen Beispielquelltexten fällt auf, dass dort als Adresse ein Wert über 127 steht, obwohl nur eine 7-Bit-Adressierung verwendet wird - selbst im Kompass-Datenblatt steht geschrieben, der Kompass hätte die Adresse 192. Da die Datenrichtung mit der Slave-Adresse als ein Byte auf den Bus gelegt wird, wird sie oftmals direkt mit in den Parameter übernommen. In diesem Datenbyte werden die höchstwertigen Bits für die Adresse verwendet und die Datenrichtung als letztes Bit angehängt; von daher ist das Datum doppelt so groß wie die eigentliche I^2C -Adresse - und im Lesemodus nochmals um eines erhöht. Die richtige Adresse des Kompassmoduls ist somit die 96, das Datenbyte für Schreibmodus lautet 192, das Datenbyte für Lesemodus lautet 193.

Datenübertragung

Nachdem nun der Slave angesprochen und die Datenrichtung festgelegt wurde, findet der eigentliche Datenaustausch statt. Der Master sorgt für die Impulse auf der SCK-Leitung und je nach Modus legen er oder der Slave die Daten auf die SDA-Leitung. Natürlich ist im Master Receiver-Modus zu beachten, dass der Master auch das ACK für die erhaltenen Daten zu senden hat.

Hat der Slave verschiedene Möglichkeiten, Daten zu versenden (z.B. durch verschiedene Register), so muss ihm zunächst mitgeteilt werden, welche Daten er senden soll. Wie schon oben erwähnt, bietet es sich dazu an, zunächst im Schreibmodus auf den Slave zuzugreifen und ihm als Datum die Anweisungen für den darauf folgenden Lesezugriff zu übermitteln.

Beispielsweise hat der Kompass ein Register für Firmware, ein Register für die aktuelle Ausrichtung auf Byte-Wert normiert, zwei Register für die aktuelle Ausrichtung in Dezimalgrad (3599 entspricht 359,9 Grad, hierbei werden dann schon zwei Bytes benötigt, aber der zurückgegebene Wert ist deutlich intuitiver) sowie noch Register für Kalibrierung und andere Informationen. Um die aktuelle Ausrichtung als Dezimalgrad zu erhalten, wird zunächst eine 2 (Auswahl des Registers) auf den Kompass geschrieben und anschließend eine lesende Datenübertragung gestartet.

```
SendeStartBedingung
SendeAdresseMitSchreibeModus
SchreibeDaten(2)
SendeRepeatedStart
SendeAdresseMitLeseModus
LeseDaten(2 Byte)
SendeStoppBedingung
```

Listing 4.29: Kommunikation mit dem Kompassmodul

Welche “Kommandos” vom Gumstix an den Atmel welche Funktion haben beziehungsweise welche Register zum Auslesen anfordern, wurde bisher noch nicht definiert.

Clock Stretching

An einem I^2C -Bus müssen Komponenten mit unterschiedlicher Geschwindigkeit arbeiten können, daher musste eine Möglichkeit gefunden werden, den Bus für langsamere Teilnehmer anzuhalten. Diese Möglichkeit findet sich beispielsweise im “Clock Stretching”:

Wenn der Slave an einem gewissen Punkt in der Übertragung nicht bereit für das nächste Datenbyte ist, drückt er die SCK-Leitung auf GND, bis er wieder bereit ist. Somit wird verhindert, dass der Master den nächsten SCK-Impuls sendet (zumindest, wenn der Master Clock Stretching unterstützt).

Der ATmega32 unterstützt Clock Stretching. Wenn beispielsweise ein Datenpaket eingetroffen ist, wird ein entsprechendes Flag gesetzt. Die SCK-Leitung wird erst dann wieder freigegeben, wenn das Programm es explizit erzwingt. Der Programmierer hat somit ausreichend Zeit, sämtliche Operationen und Vorbereitungen vorzunehmen, die das Datenpaket

verlangt.

Da selbst das Auslesen des Datenpaketes durch den Softwarefluss geschehen muss, könnte selbst *die Abfrage des I²C-Busses an sich* verzögert werden.

Natürlich gilt das nur in der Theorie! In der Praxis sollte die I²C-Kommunikation zügig abgewickelt werden. Zu diesem Zweck können I²C-Ereignisse auch Interrupts auslösen.

5 Ausblick

Das zweite, dritte und vierte Kapitel beschreibt, was die Projektgruppe RAPTOR sich im Wintersemester 2007/2008 erarbeitet hat. Dieser Abschnitt befasst sich mit den Aufgaben, die für das zweite, und damit abschließende, Semester der Projektgruppe geplant sind. Bisher hat die Gruppe vor allem die Hardware des Roboters erstellt. Dieser Bereich ist größtenteils abgeschlossen. Es müssen unter anderem noch einige Arbeiten an der Karosserie fertig gestellt werden. Eventuelle Verbesserungen bzw. Änderungen bei der Programmierung der Sensoren, des Mikrocontrollers und bei der Hardware der Hauptplatine sind nicht auszuschließen. Weiterhin wird noch eine Webcam benötigt. Die effektive Auflösung sollte 640x480 betragen, ohne dass sie dabei mit Hilfe von Software interpoliert wird. Dies ist Mindestvoraussetzung, um den Verschmutzungsgrad des Bodens erkennen zu können. Zusätzlich wird die Motorsteuerung verbessert werden, so dass der Motor beispielsweise zusätzlich mit Hilfe von Interrupts gesteuert werden kann. In dringenden Fällen, wie Kontakt mit einem Hindernis oder ein Abgrund, soll der Motor sofort zum Stillstand gebracht werden. Geplant ist, in den Anfangswochen des neuen Semesters den Hardwarebereich abzuschließen. Für das zweite Semester liegt der Fokus auf der Softwareentwicklung des Staubsaugerroboters. Demnach wird sich im Sommersemester 2008 die Hardwaregruppe der Softwaregruppe anschließen, um diese zu unterstützen. Die folgenden Aufgaben sind hierbei vorgesehen:

Die API des Simulators wird fertiggestellt. Zusätzlich wird die API für den Gumstix programmiert. Für das Verfahren der Positionsbestimmung des autonomen Roboters durch die Beobachtung seiner Räder, wird die Odometrie programmiert. Die Bahnplanung steht ebenfalls aus. Sie dient der Berechnung einer optimalen Route, die jeden Punkt im Raum mindestens einmal erreicht. Ferner soll ein Webinterface erstellt werden um eine möglichst intuitive Bedienung des Staubsaugerroboters zu ermöglichen. Somit können alle wichtigen Einstellungen über einen Webbrowser vorgenommen werden. Der wohl aufwändigste Programmierteil wird die Sensordatenfusion sein. Diese ist für die Vereinigung der Informationsausgaben aller Sensoren zuständig. Des Weiteren muss die Software des ATmega32 fertig gestellt werden. Da bisher das Controllerboard zwar in der Belegung, aber noch nicht real vorliegt, wurden die Beispielprogramme noch nicht auf die endgültige Pinbelegung angepasst.

Für den Wettbewerb, der in Abschnitt 1.2.2 beschrieben wurde, steht eine zusätzliche Funktion aus, der sogenannte Follow-Mode. Er dient dazu, dem Menschen zu folgen. Die Zusatzfunktion ermöglicht dem Staubsaugerroboter beispielsweise ein Tablett mit Getränken zu fahren. Da der Wettbewerb nicht Teil der Projektgruppe ist, sondern auf freiwilliger Basis beruht, wird diese Programmierung eine niedrige Priorität haben.

A Hardwarekomponenten

A.1 Skizzen der Karosserie

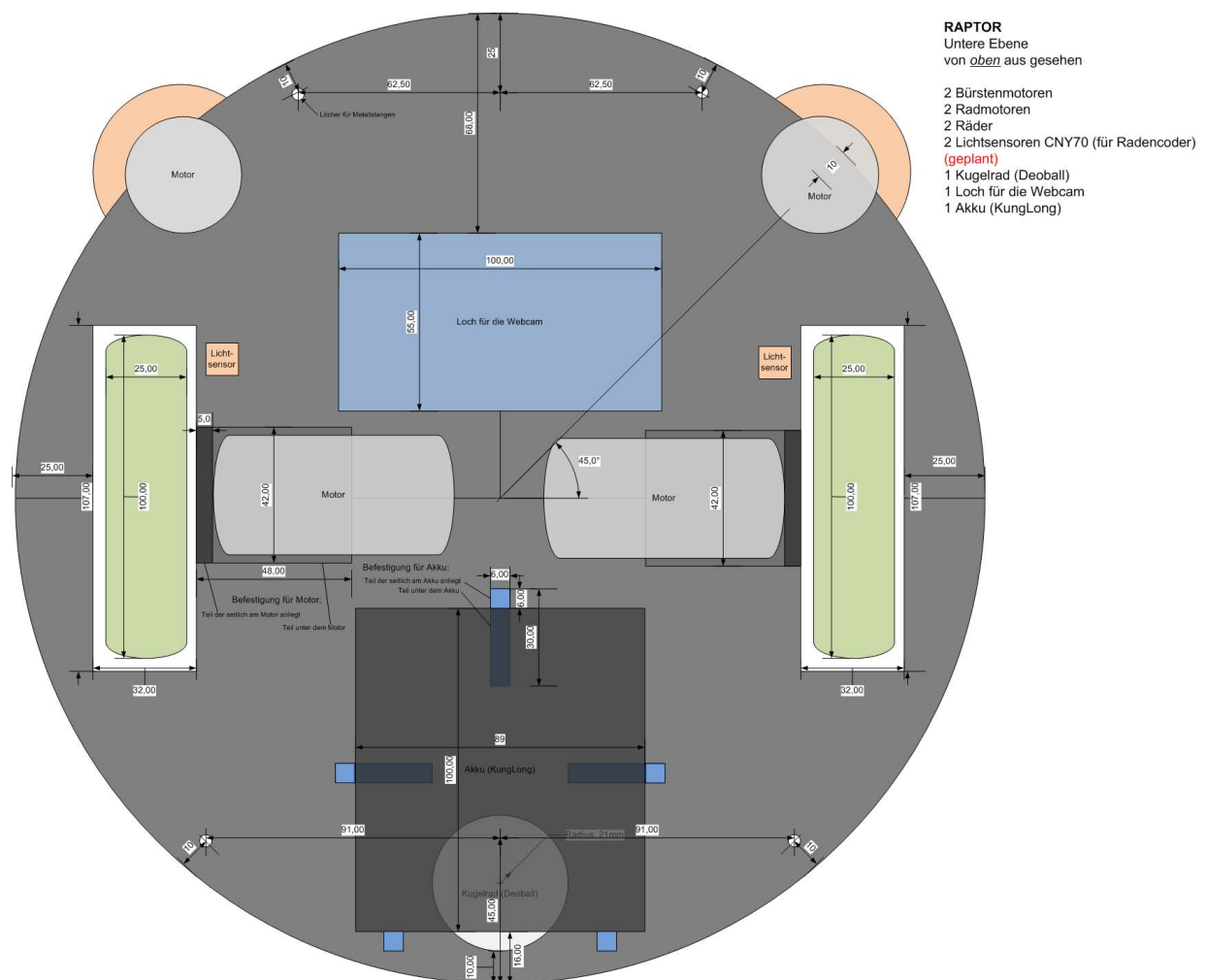


Abbildung A.1: Untere Ebene von oben aus gesehen

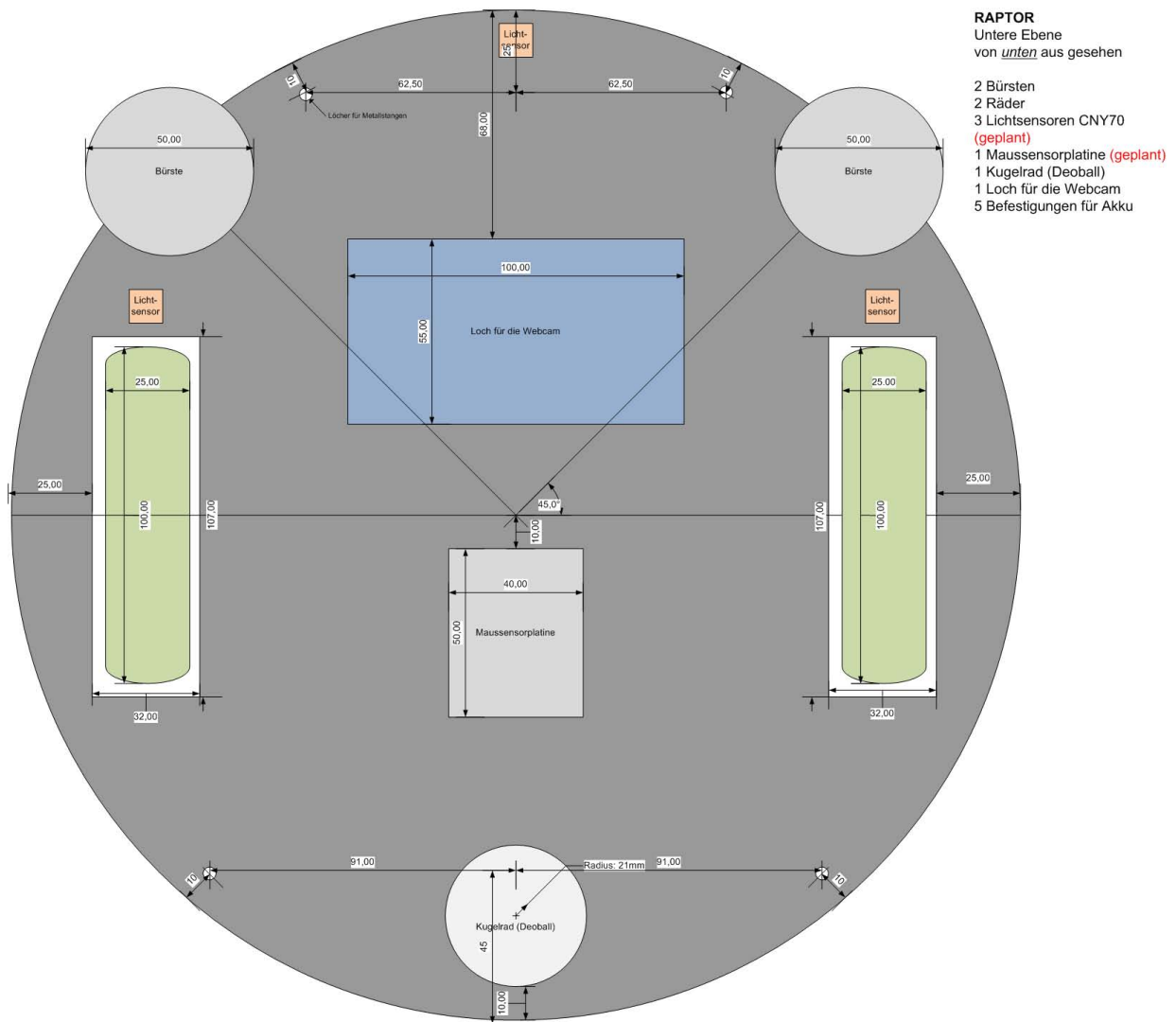


Abbildung A.2: Untere Ebene von unten aus gesehen

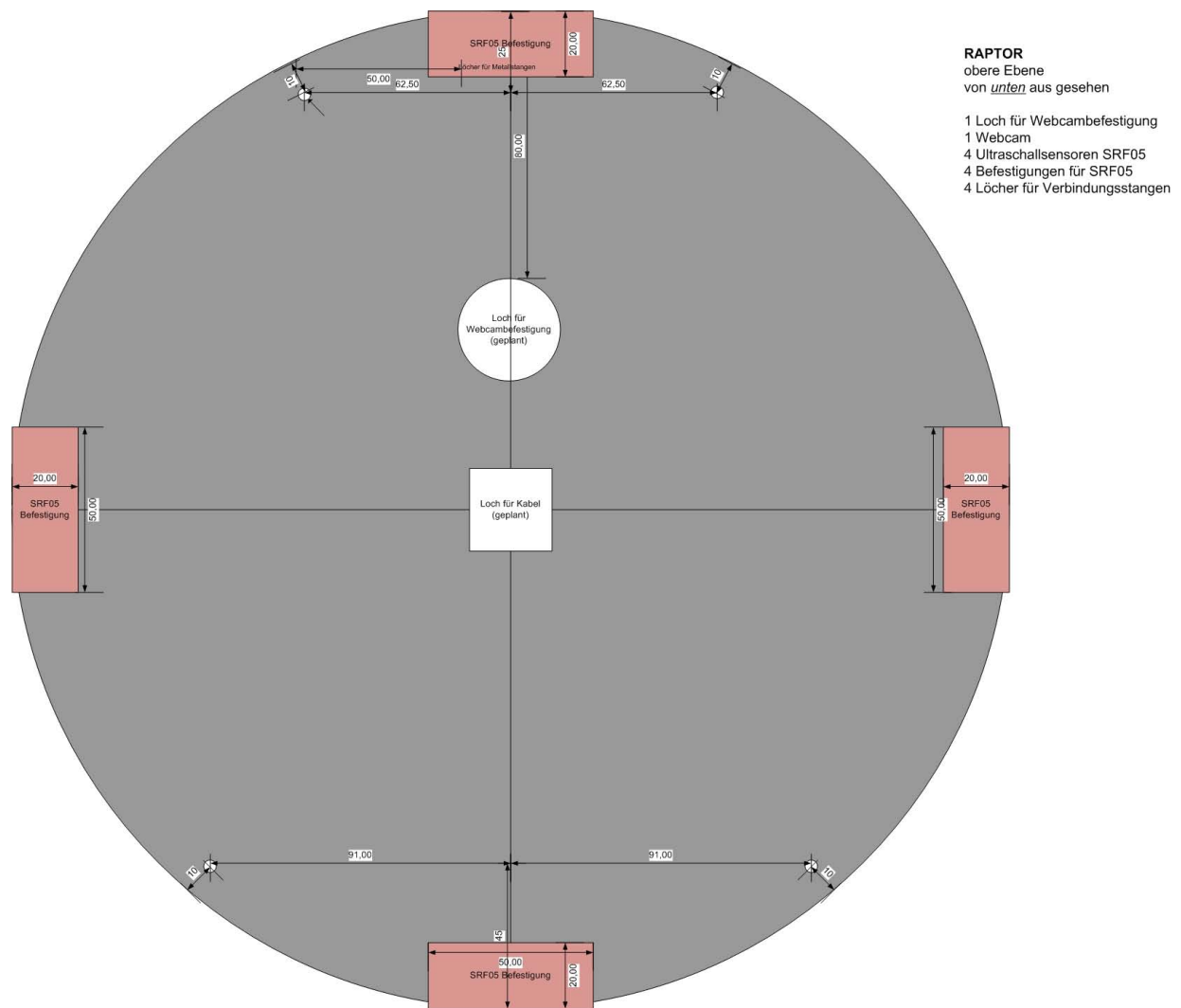


Abbildung A.3: Obere Ebene von unten aus gesehen

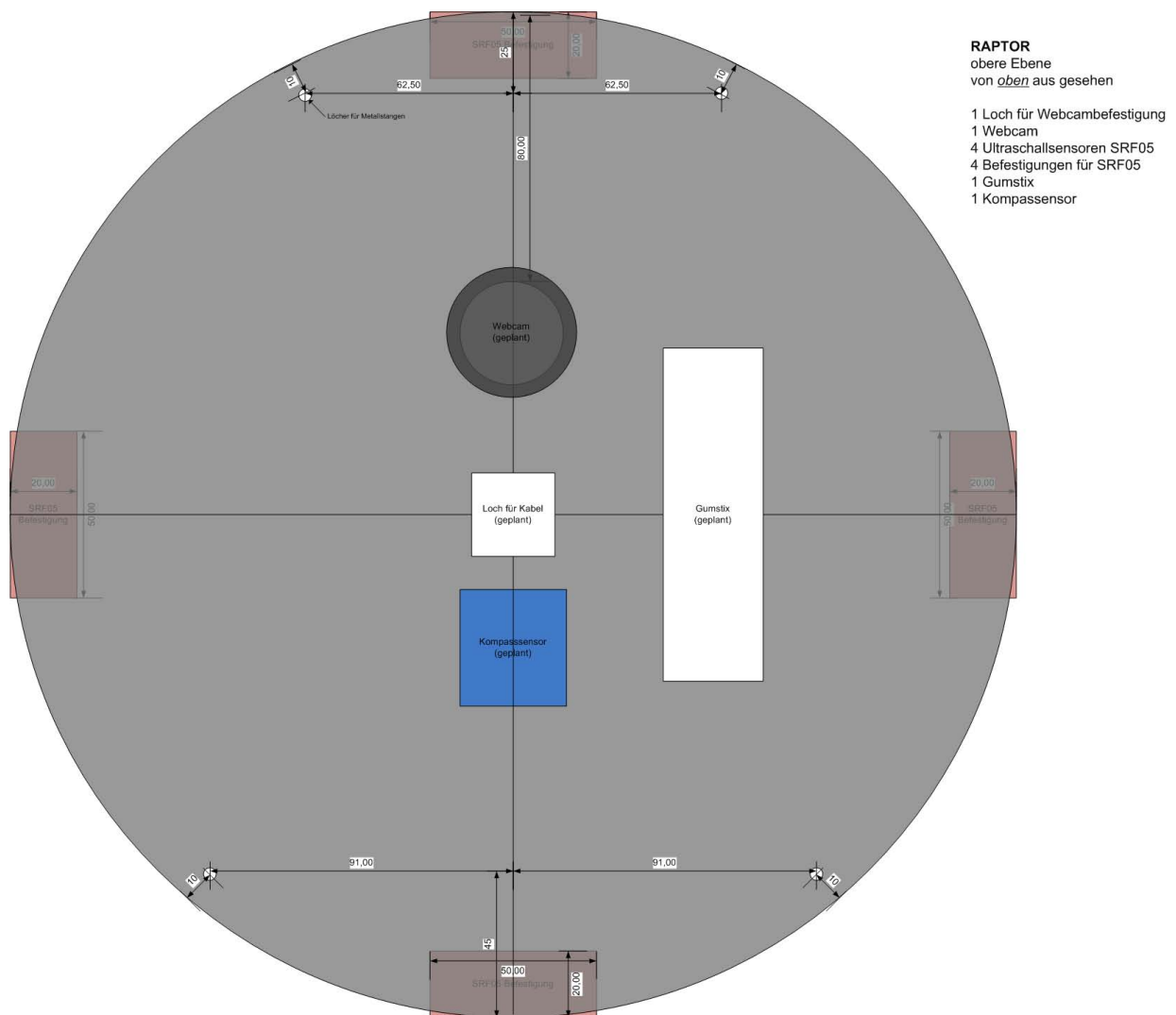


Abbildung A.4: Obere Ebene von oben aus gesehen

A.2 Schaltpläne

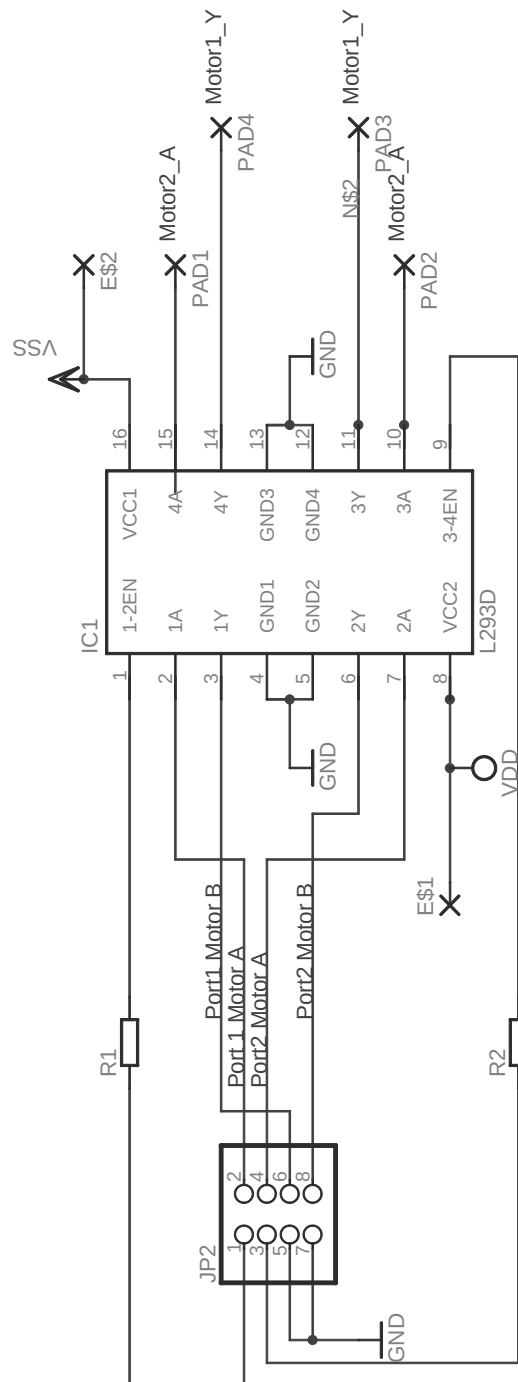


Abbildung A.5: Testschaltung der Motoren

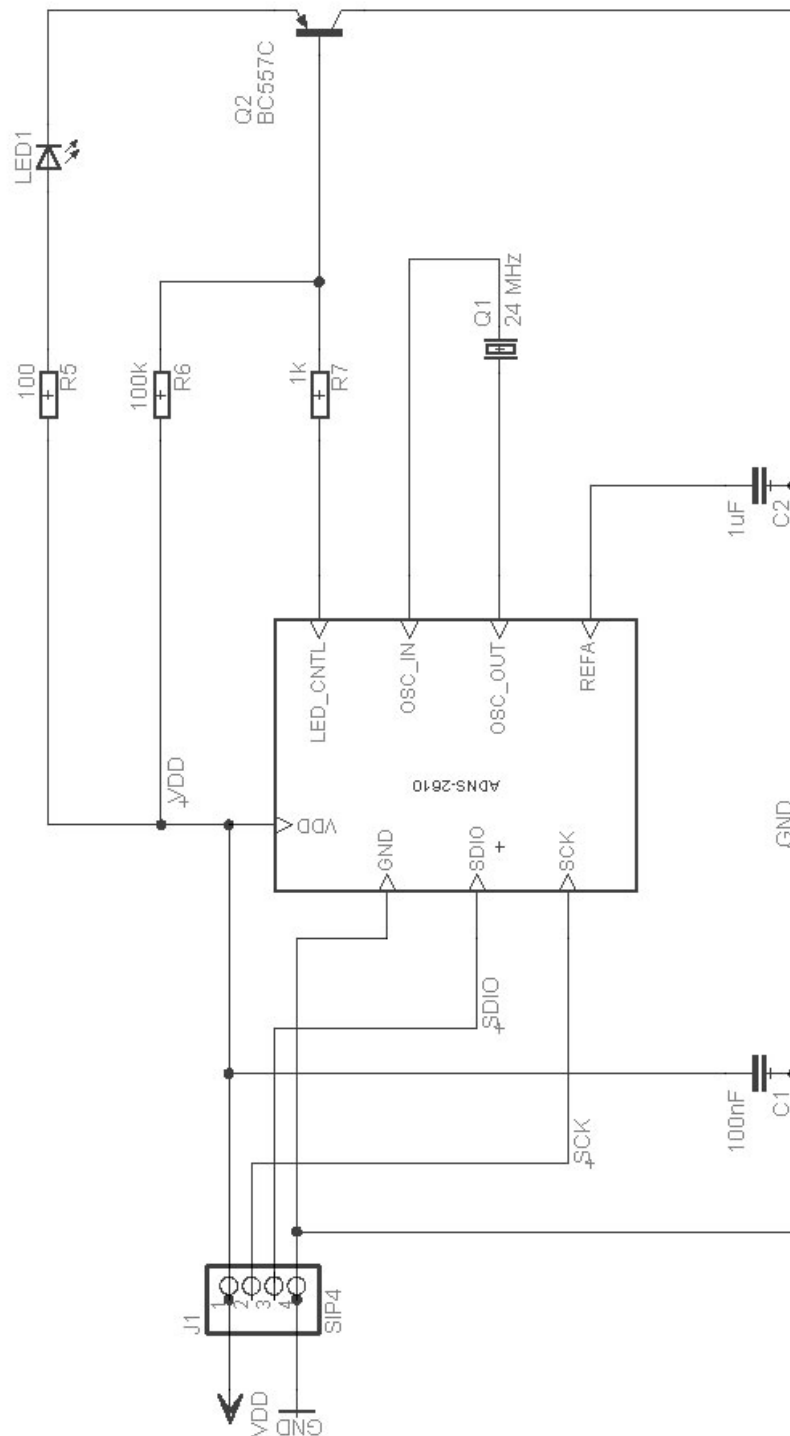


Abbildung A.6: Schaltung der Maussensorplatine

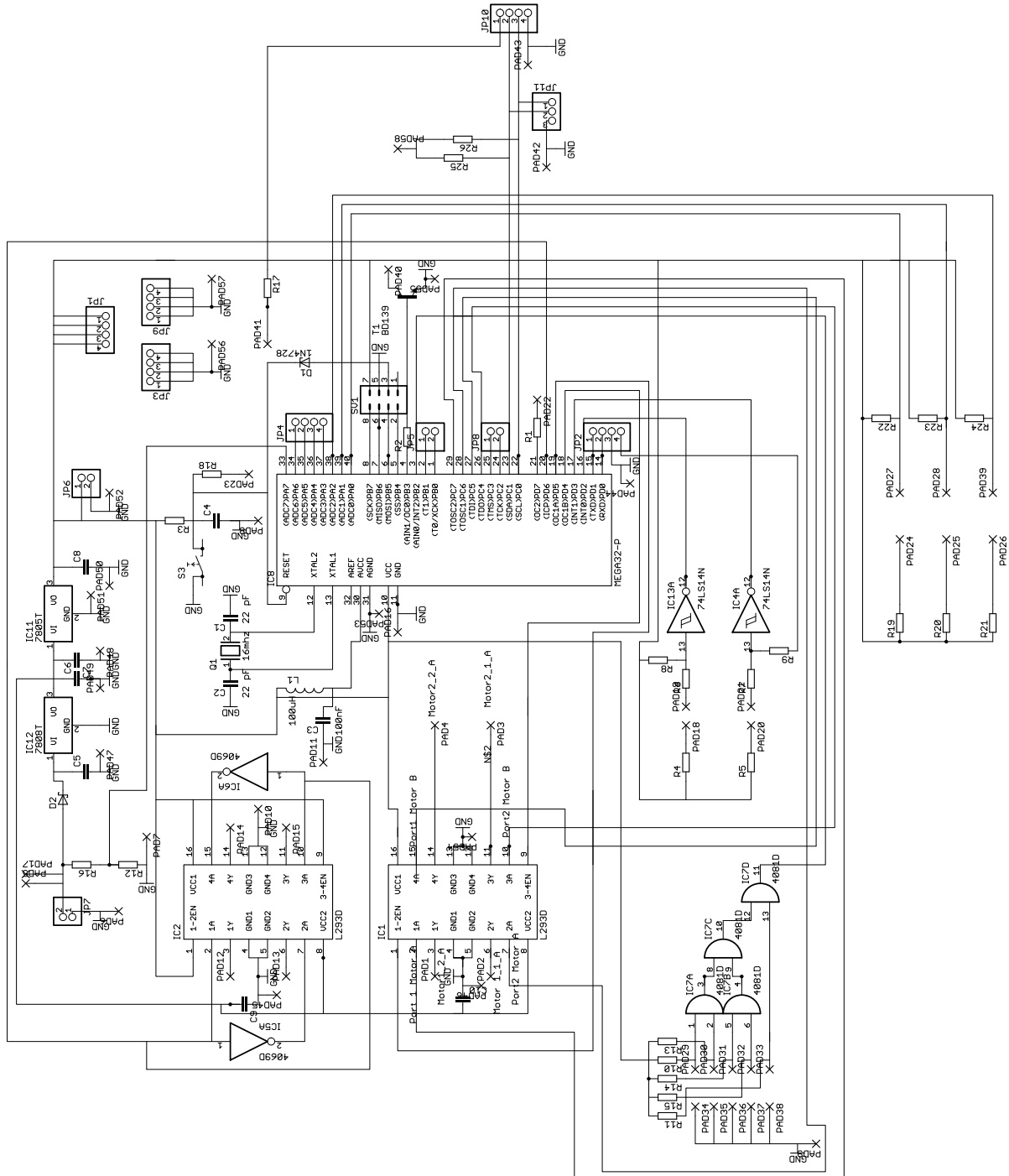


Abbildung A.7: Schaltplan der Hauptplatine

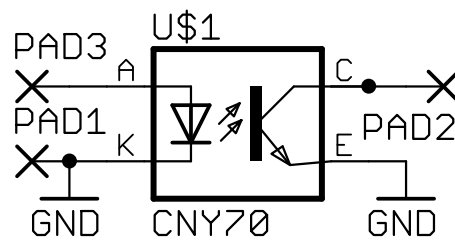


Abbildung A.8: Schaltplan der Platine für den Lichtsensor

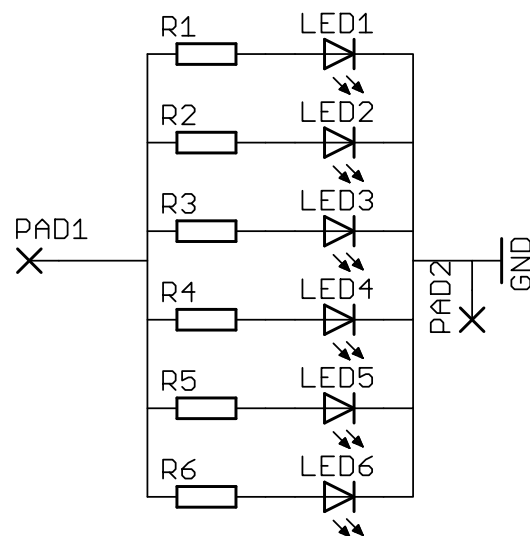


Abbildung A.9: Schaltplan der Platine für die Bodenbeleuchtung

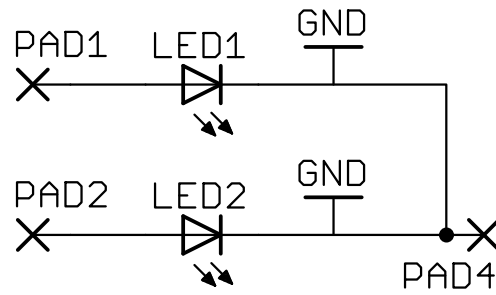


Abbildung A.10: Schaltplan der Platine für die Leuchtdioden Betriebszustand und Akkumulatorzustand

A.3 Platinenlayouts

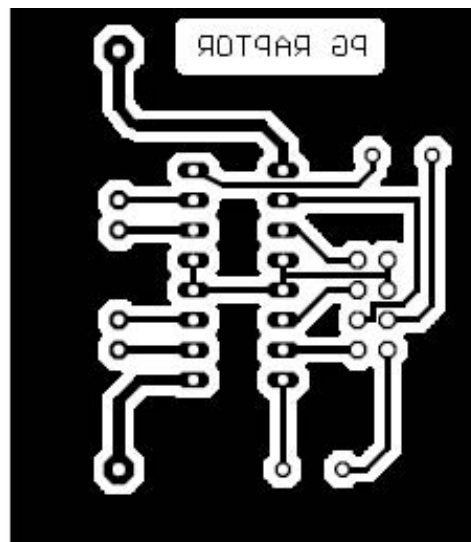


Abbildung A.11: Testplatine der Motoren

B Quelltexte

B.1 Beispielprogramme für die BreakoutBox

```
1  int main(int argc, char* argv[])
2  unsigned short myResult;
3  // unsigned short ist Vorgabe der Library
4  myResult=0;
5  cbDConfigPort(0,FIRSTPORTA,DIGITALIN);
6  /*
7   * erstes PMD-Board am System: 0
8   * Klemme 21 am ersten digitalen Port des Geraetes:
9   *     FIRSTPORTA
10  * Verwendung als Eingang: DIGITALIN
11  */
12
13  while(1) {
14      cbDBitIn(0,FIRSTPORTA,0,&myResult);
15      /*
16       * erstes PMD-Board am System: 0
17       * Klemme 21 am ersten digitalen Port des Geraetes:
18       *     FIRSTPORTA
19       * Klemme 21 ist erster Pin am Port: 0
20       * Uebergabe als Zeiger, damit Wert veraenderbar
21       */
22
23      std::cout << "Hallo_Welt,_Das_Ergebnis_ist_"<<
24      myResult << std::endl;
25      Sleep(1000);
26      /*
27       * Pause zwischen den Messungen: 1 Sekunde
28       * Methode stammt aus windows.h
29       */
30  }
31  return 0;
32 }
```

Listing B.1: Digitaler Eingang

```

1  int main(int argc, char* argv[]){
2      cbDConfigPort(0, FIRSTPORTA, DIGITALOUT);
3      unsigned short myResult=0;
4      while(1){
5          myResult=1-myResult;
6          cbDBitOut(0, FIRSTPORTA, 0, myResult);
7          std::cout << "Hallo_Welt, _Jetzt_ist_" <<
8              (myResult?"an":"aus") << std::endl;
9          Sleep(500);
10     }
11 }
```

Listing B.2: Digitaler Ausgang

```

1  int main(int argc, char* argv[]){
2      cbDConfigPort(0, FIRSTPORTA, DIGITALIN);
3      cbDConfigPort(0, FIRSTPORTB, DIGITALOUT);
4      unsigned short myResult;
5      myResult=0;
6      while(1){
7          cbDBitIn(0, FIRSTPORTA, 0, &myResult);
8          cbDBitOut(0, FIRSTPORTA, 8, myResult);
9          std::cout << "Hallo_Welt, _jetzt_ist_" <<
10             (myResult?"an":"aus") << std::endl;
11             Sleep(1000);
12     }
13 }
```

Listing B.3: Ein- und Ausgang

```

1  int main(int argc, char* argv){
2      clock_t startzeit;
3      clock_t endezeit;
4      /*
5       * clock_t ist der vorgegebene Datentyp
6       *           der clock-Methode
7       * Es laesst sich damit rechnen
8       *           wie mit Integer-Werten
9       */
10     cbDConfigPort(0, FIRSTPORTA, DIGITALIN);
11     cbDConfigPort(0, FIRSTPORTB, DIGITALOUT);
```

```
12  unsigned short myResult;
13  myResult=0;
14  while(1){
15      Sleep(1000);
16      /*
17       * Pause von 1 Sekunde vor jeder Messung
18       */
19      cbDBitOut(0,FIRSTPORTA,8,1);
20      cbDBitOut(0,FIRSTPORTA,8,0);
21      /*
22       * Die Impulsbreite von 10 Mikrosekunden
23       * wird durch Latenz automatisch erreicht.
24       */
25      while (myResult==0)
26          {cbDBitIn(0,FIRSTPORTA,0,&myResult);}
27      /*
28       * Bei Verpassen des Impulses: Endlosschleife
29       */
30      startzeit=clock();
31      /*
32       * clock() stammt aus time.h
33       */
34      while (myResult==1)
35          {cbDBitIn(0,FIRSTPORTA,0,&myResult);}
36      /*
37       * Bei Verpassen des Impulses: Endlosschleife
38       */
39      endezeit=clock();
40      std::cout << "Zeiten:_" << startzeit <<
41          ", " << endezeit << "_Differenz:_" <<
42      (endezeit-startzeit) << std::endl;
43  }
44 }
```

Listing B.4: Auslesen des Ultraschallsensors - nur teilweise funktionsfähig

B.2 Basic-Beispielprogramme für den ATMega32

```
1  $regfile = "m32def.dat"
2  $framesize = 32
3  $swstack = 32
4  $hwstack = 32
```

```

5 $crystal = 1000000
6
7 Do
8 Loop
9
10 End

```

Listing B.5: Ein Nullprogramm

```

1 $regfile = "m32def.dat"
2 $framesize = 32
3 $swstack = 32
4 $hwstack = 32
5 $crystal = 1000000
6
7 Config Portc.0 = Output
8
9 Do
10   Portc.0 = 1
11   Waitms 100
12   Portc.0 = 0
13   Waitms 100
14 Loop
15
16 End

```

Listing B.6: Leuchtdiode blinken lassen

```

1 $regfile = "m32def.dat"
2 $framesize = 32
3 $swstack = 32
4 $hwstack = 32
5 $crystal = 1000000
6
7 Config Portc.0 = Output
8 Config Pina.7 = Input
9 Porta.7 = 1
10
11 Do
12   Portc.0 = 1
13   If Pina.7 = 1 Then
14     Waitms 100
15     Portc.0 = 0
16     Waitms 100

```

```
17  Else
18      Waitms 500
19      Portc.0 = 0
20      Waitms 500
21  End If
22 Loop
23 End
```

Listing B.7: Tastereingabe verarbeiten

```
1  $regfile = "m32def.dat"
2  $framesize = 32
3  $swstack = 32
4  $hwstack = 32
5  $crystal = 16000000
6  $baud = 9600
7
8  Config Portc.0 = Output
9  Config Pina.7 = Input
10 Porta.7 = 1
11
12 Do
13     Print "aus"
14     Portc.0 = 1
15     If Pina.7 = 1 Then
16         Waitms 100
17         Print "an"
18         Portc.0 = 0
19         Waitms 100
20     Else
21         Waitms 500
22         Print "an"
23         Portc.0 = 0
24         Waitms 500
25     End If
26 Loop
27 End
```

Listing B.8: serielle Kommunikation

```
1  $regfile = "m32def.dat"
2  $framesize = 32
3  $swstack = 32
4  $hwstack = 32
```

```

5 $crystal = 16000000
6 $baud = 9600
7
8 Config PORTC.0 = Output
9 Config PORTC.1 = Output
10 Config PORTA.7 = Input
11
12 Dim Zeitmessung As Word
13 Dim Ledstatus As Byte
14 Ledstatus=0
15 Do
16     PORTC.0 = Ledstatus
17     Ledstatus = 1 - Ledstatus
18
19     PORTC.1 = 0
20     Pulseout PORTC , 1 , 40
21     Pulsein Zeitmessung, PINA , 7 , 1
22     Zeitmessung = Zeitmessung * 100
23     Zeitmessung = Zeitmessung / 58
24     Print "Abstand:_ " ; Zeitmessung ; "_cm"
25     Waitms 50
26 Loop
27 End

```

Listing B.9: Auslesen des Ultraschall-Sensors

B.3 C-Beispielprogramme für den ATMega32

```

1 int main(){
2     while(1){
3         // Endlosschleife
4     }
5     return 0; // nur der Schönheit wegen
6 }
7 /*
8  * Alternativ:
9  *
10  * int main(void){
11  *     for (;;) { /*nothing*/ }
12  *     return 0;
13  * }
14  */

```

Listing B.10: Wieder ein Nullprogramm

```
1 #include <avr/io.h>
2 int main(void) {
3     DDRC = 0xff;
4     PORTC = 0x00;
5     for (;;) { /*nothing*/ }
6     return 0;
7 }
```

Listing B.11: Leuchtdiode anschalten

```
1 #include <avr/io.h>
2 int main(void)
3     DDRC=0xff;
4     DDRA=0x00;
5     for (;;) {
6         if ((PIN_A & 0x40) == 0) {
7             PORTC=0x00;
8         } else {
9             PORTC=0x01;
10        }
11    }
12    return 0;
13 }
```

Listing B.12: Eingang auslesen

```
1 // #define F_CPU 16000000L
2 #define BAUDEXT 9600L
3 #define BAUDINT (BAUDEXT/17/BAUDEXT -1)
4 int uart_init() {
5     UCSRB |= (1<<TXEN);
6     UBRRH = BAUDINT >> 8;
7     UBRRL = BAUDINT & 0xff;
8 }
9
10 int uart_putc(unsigned char c) {
11     while (!(UCSRA & (1<<UDRE))) {}
12     /* und warten und warten und warten */
13     UDR = c;
14     return 0;
15 }
```

```

16
17 int uart_puts(char *s) {
18     while (*s) {
19         uart_putc(*s);
20         s++;
21     }
22     return 0;
23 }

```

Listing B.13: Serielle Ausgaben

```

1 #define MAX_WAIT_MS (262/(F_CPU/1000000))
2 void sleepms(int ms) {
3     while (ms>MAX_WAIT_MS) {
4         _delay_ms(MAX_WAIT_MS); ms-=MAX_WAIT_MS;
5     }
6     _delay_ms(ms);
7 }

```

Listing B.14: Wrapper für Warteschleifen

```

1 void TWIerror(int statuscode) {
2     switch (statuscode & 0xf8) {
3         case TW_REP_START: uart_puts("REP_START");
4         break;
5         case TW_MT_SLA_NACK: uart_puts("MT_SLA_NACK");
6         break;
7         case TW_MT_DATA_NACK: uart_puts("MT_DATA_NACK");
8         break;
9         case TW_MR_SLA_ACK: uart_puts("MR_SLA_ACK");
10        break;
11        case TW_MR_DATA_NACK: uart_puts("MR_DATA_NACK");
12        break;
13        case TW_MR_SLA_NACK: uart_puts("MR_SLA_NACK");
14        break;
15        case TW_SR_SLA_ACK: uart_puts("SR_SLA_ACK");
16        break;
17        case TW_SR_DATA_ACK: uart_puts("SR_DATA_ACK");
18        break;
19        case TW_SR_STOP: uart_puts("SR_STOP");
20        break;
21        default: {
22            uart_puts("unknown_TWI_Error");
23            break;

```

```
24     }
25 }
26 char s[3];
27 uart_puts(utoa(statuscode,s,16));
28 while (1){}; // endlosloop
29 }
30
31
32 #include <util/twi.h>
33 int main (void){
34     TWCR = (1<<TWEN) | (1 << TWEA);
35     char s[11];
36
37     uint8_t zaehler;
38     zaehler=0;
39
40     uint8_t modus;
41     //modus=1; // modus 0 sei Master
42     //modus=0; // modus 1 sei Slave
43
44     TWBR = 72;
45     // mit ein wenig Umrechnung hat man damit 100 kHz;
46     if(modus==0){ // master-Modus
47         for (;;) {
48             zaehler++;
49             uart_puts("\n\rStartsignal");
50             // Startsignal und Trigger
51             TWCR = (1 << TWEN) | (1 << TWEA) |
52                 (1 << TWSTA) | (1 << TWINT);
53             // Startsignal und Trigger
54             while (!(TWCR & (1<<TWINT))){}
55             // warte auf Anforderung
56             if ((TWSR & 0xf8) != TW_START){TWIerror(TWSR);}
57             uart_puts("_ok");
58
59             TWDR=194; // Adressauswahl
60             // TWSTA wird geloescht, Daten (Adresse) senden
61             TWCR = (1 << TWEN) | (1 << TWEA) | (1 << TWINT);
62             // TWSTA wird geloescht, Daten (Adresse) senden
63             while (!(TWCR & (1<<TWINT))){}
64             // warte auf Anforderung
65             if ((TWSR & 0xf8) != TW_MT_SLA_ACK)
66                 {TWIerror(TWSR);}
```

```

67     uart_puts("\n\rSlave_ACK");
68
69     TWDR=zaehler; // Registerauswahl
70     // Daten (Register) senden
71     TWCR = (1 << TWEN) | (1 << TWEA) | (1 << TWINT);
72     // Daten (Register) senden
73     while (!(TWCR & (1<<TWINT))){}
74     // warte auf Anforderung
75     if ((TWSR & 0xf8) != TW_MT_DATA_ACK)
76         {TWIerror(TWSR);}
77     uart_puts("\n\rData_ACK");
78     uart_puts(ltoa(zaehler,s,10));
79     // gibt die gesendeten Daten
80
81     TWCR = (1 << TWEN) | (1 << TWINT) | (1 << TWSTO);
82     uart_puts("\n\rStop_erfolgreich_gesendet\n\r");
83     sleepms(200);
84 }
85 }
86 if (modus==1){ // slave-modus
87     TWAR=194;
88     for(;;){
89         uart_puts("\n\rIch_warte");
90         while(!(TWCR & (1<<TWINT))){}
91         // warte auf Anforderung
92         if ((TWSR & 0xf8) != TW_SR_SLA_ACK)
93             {TWIerror(TWSR);}
94         uart_puts("\n\rIch_schicke_ein_ACK");
95         sleepms(200);
96
97         // ich bin bereit
98         TWCR= (1<<TWEN) | (1 << TWEA) | (1 << TWINT);
99         // ich bin bereit
100        while(!(TWCR & (1<<TWINT))){}
101        // warte auf Anforderung
102        if ((TWSR & 0xf8) != TW_SR_DATA_ACK)
103            {TWIerror(TWSR);}
104        uart_puts("\n\rDaten");
105        zaehler=TWDR;
106        uart_puts(ltoa(zaehler,s,10));
107        // gibt die empfangenen Daten
108
109        // ich warte noch auf stopp

```

```
110     TWCR= (1<<TWEN) | (1 << TWEA) | (1 << TWINT);
111     // ich warte noch auf stopp
112     while(!(TWCR & (1<<TWINT))){}
113     // warte auf Anforderung
114     if ((TWSR & 0xf8) != TW_SR_STOP)
115         {TWIerror(TWSR);}
116     uart_puts("\n\rStopp_erhalten");
117
118     // ich bin wieder bereit
119     TWCR= (1<<TWEN) | (1 << TWEA) | (1 << TWINT);
120     // ich bin wieder bereit
121 }
122 }
123 return 0;
124 }
```

Listing B.15: TWI-Kommunikation

```
1 volatile uint16_t zeiter;
2 /*
3  * zeiter muss als "volatile" definiert sein,
4  * damit die Interruptroutine die
5  * Variable aendern kann.
6  */
7
8 int main(){
9     sei();
10    // Programm fuer Ultraschall
11    // Ich verwende Timer0, normal mode (WGM01=WGM00=0)
12    // und der OC0-Pin ist nicht verbunden (COM01=COM00=0)
13    TIMSK |= (1<<TOIE0);
14    /*
15     * aktiviert einen Interrupt-Modus
16     * Da Counter noch nicht laeuft,
17     * werden keine Interrupts ausgeloeset
18     *
19     * Die restlichen Register enthalten
20     * schon beim Start die Werte,
21     * die wir wuenschen
22     */
23
24    while(1){
25        // Messung
```

```

26 // DDR Hi: Pin ist Ausgang
27 // Ultraschall ist angeschlossen an PB0
28
29 /*
30  * Vor jeder Messung:
31  * Timer stoppen und 0 setzen
32  */
33 zeiter=0;
34 TCNT0=0;
35
36 /*
37  * Impuls rausjagen
38  * Dabei wird der Pin kurzfristig
39  * auf Ausgang geschaltet
40  */
41 DDRB |= (1<<PB0);
42 PORTB |= (1<<PB0);
43 _delay_us(10);
44 PORTB &=~ (1<<PB0);
45 DDRB &=~ (1<<PB0);
46
47 /*
48  * Warte, bis der Impuls kommt
49  * (steigende Flanke)
50  */
51 while ((PINB & (1<<PB0))==0) {}
52 /*
53  * Nach Erkennung der Flanke:
54  * Sofort: Timer starten!
55  * (Durch Zuweisung von Taktquelle)
56  */
57 TCCR0 |= (2<<CS00);
58
59 /*
60  * Warte, bis Impuls geht
61  * (fallende Flanke)
62  * Waehrend dieser Schleife werden
63  * die Interrupts ausgeloeset
64  */
65 while (PINB & (1<<PB0)) {}
66 /*
67  * Sofort: Timer stoppen
68  * um weiter Interrupts

```

```
69     * zu vermeiden
70     */
71     TCCR0 &=~ (7<<CS00);
72
73     /*
74     * Timer auslesen
75     * Jeder Overflow (in zeiter gezaehlt)
76     * entspricht einem Zaehlvorgang von
77     * 0 bis 255. <<8 entspricht *256
78     */
79     zeiter=(zeiter<<8)+TCNT0;
80
81     /*
82     * Umrechnung von Pulsbreite
83     * in Zentimeter
84     * Siehe Datenblatt SRF05
85     */
86     uart_puts(ultoa(zeiter,s,10));
87     uart_puts("*500_nS\n\r");
88     zeiter=zeiter/2;
89     uart_puts(ultoa(zeiter,s,10));
90     uart_puts("_uS\n\r");
91     zeiter=zeiter/58;
92     uart_puts(ultoa(zeiter,s,10));
93     uart_puts("_Zentimeter\n\r");
94     uart_puts("\n\r");
95     sleepms(200);
96 }
97 }
98 /*
99 * Interrupt-Routine:
100 * moeglichst kurz:
101 * nur ein Inkrement
102 */
103 ISR(TIMER0_OVF_vect) {
104     zeiter=zeiter+1;
105 }
```

Listing B.16: Impulsmessung

```
1 #define PinSCK 0
2 #define PinSDA 1
3
```

```

4 // Maussensor
5 // SCK landete an PIN40 (porta.0)
6 // SDIO/SDA landete an PIN39 (porta.1)
7 void pan_writeByte(unsigned char data){
8     /*
9      * Schickt ein Byte an den Sensor
10     * Kann auch die Adresse eines
11     * Registers sein, welches
12     * gelesen werden soll
13     */
14     int8_t i;
15     DDRA |= (1<<PinSDA); // SDA auf Output
16     for (i=7; i>=0; i--){
17         PORTA &= ~(1<<PinSCK);
18         //SCK auf Low, Daten vorbereiten
19
20         if(data&(1<<i)){ //Bit rausschieben
21             PORTA |= (1<<PinSDA);
22         }else{
23             PORTA &= ~(1<<PinSDA);
24         }
25
26         PORTA |= (1<<PinSCK);
27         // SCK =1
28         // Sensor uebernimmt auf steigender Flanke
29
30         _delay_us(1);
31         //Sensor Zeit lassen um Bit zu holen
32     }
33     DDRA &= ~(1<<PinSDA);
34     //HI-Z state
35     PORTA &= ~(1<<PinSDA);
36 }
37
38 unsigned char pan_readByte(void){
39     /*
40     * Empfaengt ein Byte vom Sensor
41     * Register muss vorher
42     * adressiert worden sein
43     */
44     int8_t i;
45     unsigned char data=0;
46     _delay_us(3);

```

```
47 // Sensor Zeit lassen um die Daten
48 // aus dem Register zu holen
49 for (i=7; i>-1; i--){
50     PORTA &= ~ (1<<PinSCK);
51     // SCK =0 Sensor bereitet Daten
52     // auf fallender Flanke vor!
53
54     _delay_us(1);
55     //Sensor kurz zeit lassen
56
57     PORTA |= (1<<PinSCK);
58     // SCK =1 Daten lesen auf steigender Flanke
59
60     if(PINA & (1<<PinSDA)){//BIT einlesen
61         data |= (1<<i);
62     }else{
63         data &=~ (1<<i);
64     }
65 }
66 return data;
67 }
68
69 void pan_write(unsigned char adr, unsigned char data){
70     /*
71     * Verbindet Adressieren und
72     * Beschreiben des Registers
73     */
74     adr|=(1<<7);
75     pan_writeByte(adr);
76     //rl MSB muss 1 sein fuer Write Operation
77     pan_writeByte(data);
78 }
79
80 unsigned char pan_read(unsigned char adr){
81     /*
82     * Verbindet Adressieren und
83     * Lesen des Registers
84     */
85     pan_writeByte(adr);
86     return pan_readByte();
87 }
88
89 void pan_init(void) {
```

```

90  /*
91  * Initialisierung
92  * setzt Datenrichtungen,
93  * legt 5V-Pegel an
94  * Init-Befehle an Sensor
95  */
96  DDRA = 0b00000011;
97  // SCK und SDA auf Output
98  PORTA |= (1<<PinSDA) | (1<<PinSCK);
99  // SCK und SDA auf high
100 // hier muessen bei Umstellung auf PAN101
101 // die entsprechenden Register gesetzt werden
102 //Reset PAN3101
103 pan_write(0x00,0x80);
104 // kein Sleep modus
105 pan_write(0x00,0x01);
106 }
107
108 int main (void){
109     pan_init();
110     int8_t gelesen;
111     int16_t x;
112     int16_t y;
113     x=0;
114     y=0;
115     for(;;){
116         // Verschiebung und Positionierung
117         gelesen=pan_read(0x03);
118         uart_puts("\n\r\n\rVerschiebung_X:");
119         uart_puts(itoa(gelesen,s,10));
120         x+=gelesen;
121
122         gelesen=pan_read(0x02);
123         uart_puts("___Verschiebung_Y:");
124         uart_puts(itoa(gelesen,s,10));
125         y+=gelesen;
126
127         uart_puts("\n\rX-Koord:_");
128         uart_puts(ltoa(x,s,10));
129         uart_puts("___Y-Koord:_");
130         uart_puts(ltoa(y,s,10));
131
132         sleeps(200);

```

```
133     }
134     return 0;
135 }
136 /* Alternativ: komplettes Kamerabild auslesen
137 // Bild
138     for(;;){
139         pan_write(0x08,1);
140         for (x=1;x<=324;x++){
141             gelesen=pan_read(0x08);
142             y='#';
143             if (gelesen>SCHWELLWERT1){y='+';}
144             if (gelesen>SCHWELLWERT2){y=' ');}
145             uart_putc(y);
146             if (x % 18 == 0){uart_puts("\n\r");}else{}
147         }
148         uart_puts("\n\r\n\r\n\r\n\r\n\r\n\r");
149     }
150 */
```

Listing B.17: Kommunikation mit dem Maussensor

B.4 Motortreiber

```

1  #include <stdint.h>
2  #include <avr/io.h>
3  #include <stdlib.h>
4
5  #define MOTOR_DIR_L_PIN_1Y      PC7 // 1Y
6  #define MOTOR_DIR_L_PIN_2Y      PC6 // 2Y
7
8  #define MOTOR_DIR_R_PIN_3Y      PC5 // 3Y
9  #define MOTOR_DIR_R_PIN_4Y      PC4 // 4Y
10
11 #define MOTOR_DIR_PORT           PORTC
12 #define MOTOR_DIR_DDR            DDRC
13
14 #define PWM_DDR                   DDRD
15 #define PWM_L                     OCR1A
16 #define PWM_R                     OCR1B
17
18 uint8_t motor_left;
19 uint8_t motor_right;
20
21 int motor_init(void) {
22     MOTOR_DIR_DDR |= (1<<DDC7)
23                     | (1<<DDC6)
24                     | (1<<DDC5)
25                     | (1<<DDC4);
26
27     TCCR1A      |= (1<<WGM10)
28                     | (1<<COM1A1)
29                     | (1<<COM1B1);
30
31     TCCR1B      |= (1<<WGM12)
32                     | (1<<CS12)
33                     | (1<<CS10);
34
35     TCNT1 = 0x0000;
36
37     PWM_DDR |= (1<<PD4) | (1<<PD5);
38
39     PWM_L = 255;
40     PWM_R = 255;

```

```
41
42     return 0;
43 }
44
45 void motor_drive(uint8_t speed_left,
46                 uint8_t speed_right,
47                 bool direction){
48     motor_left = speed_left;
49     motor_right = speed_right;
50
51     PWM_L = motor_left;
52     PWM_R = motor_right;
53
54     if(direction){
55         MOTOR_DIR_PORT |= (1 << MOTOR_DIR_L_PIN_1Y);
56         MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_L_PIN_2Y);
57
58         MOTOR_DIR_PORT |= (1 << MOTOR_DIR_R_PIN_3Y);
59         MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_R_PIN_4Y);
60     }else{
61         MOTOR_DIR_PORT |= (1 << MOTOR_DIR_L_PIN_2Y);
62         MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_L_PIN_1Y);
63
64         MOTOR_DIR_PORT |= (1 << MOTOR_DIR_R_PIN_4Y);
65         MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_R_PIN_3Y);
66     }
67 }
68
69 void motor_turn_left(){
70     PWM_L = 255;
71     PWM_R = 255;
72
73     MOTOR_DIR_PORT |= (1 << MOTOR_DIR_L_PIN_2Y);
74     MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_L_PIN_1Y);
75
76     MOTOR_DIR_PORT |= (1 << MOTOR_DIR_R_PIN_3Y);
77     MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_R_PIN_4Y);
78 }
79
80 void motor_turn_right(){
81     PWM_L = 255;
82     PWM_R = 255;
83 }
```

```

84  MOTOR_DIR_PORT |= (1 << MOTOR_DIR_L_PIN_1Y);
85  MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_L_PIN_2Y);
86
87  MOTOR_DIR_PORT |= (1 << MOTOR_DIR_R_PIN_4Y);
88  MOTOR_DIR_PORT &= ~(1 << MOTOR_DIR_R_PIN_3Y);
89  }

```

Listing B.18: Motortreiber

B.5 Motortreiber der Testschaltung

```
1  /* clock Beispiel, arbeitet mit zwei wait methoden */
2
3  #include <stdio.h>
4  #include <time.h>
5  #include <windows.h>
6  #include <iostream.h>
7
8  class K
9  {
10 public :
11  //////////////////////////////////////////////////Public Methoden////////////////////////////////////
12  K(); //Konstruktor
13  ~K(); // Destruktor
14  void pwm_singal(int motor_leistung);
15  //////////////////////////////////////////////////
16  //////////////////////////////////////////////////Private Methoden & Variablen////////////////////////////////
17 private :
18  unsigned short HIGH;
19  void wait(float seconds,int i);
20  //////////////////////////////////////////////////
21  };
22  K::K()
23  {
24  HIGH = 0;
25  }
26  K::~~K() {}
27  void K::wait (float seconds,int i)
28  {
29  float endwait;
30  endwait = clock () + seconds * CLOCKS_PER_SEC ;
31
32  /*
33   * setzt den High Pegel z.B.
34   * für 0.8 Sekunden auf High
35   */
36  while (clock() < endwait) {
37  HIGH = 1;
38  cbDBitOut(0, FIRSTPORTA,0,HIGH);
39  }
40  /*
```

```

41     * die Abfrage nach 1 sagt, dass wir einen
42     * Taktzyklus < 100% haben,
43     * hier wird der Motor für die
44     * restliche Gesamtzeit auf GND gesetzt
45     */
46     if(i==1)
47     {
48         HIGH = 0;
49         /* der Motor wird ausgeschaltet, also auf GND gesetzt*/
50         cbDBitOut(0, FIRSTPORTA,0,HIGH);
51     }
52 }
53
54 /* Einstellen wie lange der High Takt in % dauert und wie
55    lang der Low Takt dauert _||_||_ */
56 void K::pwm_singal(int motor_leistung)
57 {
58     /*
59     * die Variable Periodendauer darf nicht geändert werden,
60     * muss aber noch berechnet werden
61     * (siehe Abbildung oben)
62     */
63
64     const int Gesamtzeit = 1;
65
66     /* Umrechnung in die gewünschte Taktzeitlänge */
67     float High_Takt = Gesamtzeit *
68                     float(motor_leistung) / 10;
69
70     /* der Wert wird in Millisekunden angegeben */
71     int Low_Takt = int(1000 * (Gesamtzeit - High_Takt) );
72
73     /*
74     * die IF-Abfrage verhindert,
75     * dass eine Fehlermeldung auftaucht
76     */
77     if(High_Takt <= 0)
78     {
79         printf("High_Takt_<=_0");
80         Sleep(Low_Takt);
81     }
82     else if(Low_Takt <= 0)
83     {

```

```
84  /*
85   * sagt, dass der HighPegel permanent
86   * auf 1 gesetzt ist
87   */
88  wait( High_Takt,0);
89  }
90  else
91  {
92    wait( High_Takt,1);
93    Sleep(Low_Takt);
94  }
95 }
```

Listing B.19: Header-Datei

```
1  #include <stdio.h>
2  #include "K(2).H"
3  int main(int argc, char* argv[])
4  {
5    K PWM;
6    printf ("Starting_countdown...\n");
7    while(1)
8    {
9      /*
10     * Gibt in Prozent an, wie schnell der
11     * Signalwechsel von High auf LOW erfolgt.
12     * in dem Bsp. beträgt der Signalwechsel
13     * von High auf Low bzw. Low auf High 50%
14     */
15     PWM.pwm_singal(5);
16     }
17     return 0;
18 }
```

Listing B.20: Main-Programm

Abbildungsverzeichnis

2.1	gumstix verdex XL6P [11]	14
2.2	console-vx [11]	16
2.3	netwifimicroSD EU [11]	17
2.4	Der Ultraschallsensor SRF05 [8]	18
2.5	Anschlüsse für separaten Trigger- und Echo-Pin [8]	19
2.6	Anschlüsse für gemeinsamen Trigger- und Echo-Pin [8]	20
2.7	Zeitdiagramm vom Modus 1 [8]	21
2.8	Zeitdiagramm vom Modus 2 [8]	21
2.9	Richtwirkung des Sensors [8]	22
2.10	Der Microschalter MBF5B [12]	22
2.11	Anschlüsse des Kontaktsensors [12]	23
2.12	Der Lichtsensor CNY70 [26]	24
2.13	Steckerbelegung des CNY70 [26]	24
2.14	Bestandteile des optischen Maussensors [1]	26
2.15	Aufbau und Funktionsweise der Optik des optischen Maussensors [1]	26
2.16	Pinbelegung des Kompassensors [8]	27
2.17	Ein ATMega32 in in PDIP-Bauform	31
2.18	Logitech SweetPea QuickCam Express	32
3.1	Das Steckbrett unserer Testumgebung - mit einem Testaufbau	40
3.2	Die versteckten Verbindungen am Steckbrett	41
3.3	Die versteckten Verbindungen an den Lötstiften, rechte Seite analog	42
3.4	Die "Breakout-Box" [18]	42
3.5	Der L293D Baustein [8]	49
3.6	Der L298 Baustein [8]	49
3.7	Motoransteuerungstabelle [8]	49
3.8	Bestimmung der Periodendauer [8]	50
3.9	L298 mit Freilaufdioden [8]	50
3.10	Pinbelegung des ATMega32	51
3.11	AVRISP mkII - unser Programmiergerät	52
3.12	Pinbelegung von Wannenbuchsen	53
3.13	Das kleine Adapterboard für unseren Programmieradapter	53
3.14	Verbindung zwischen ATMega und PC - nur Basis	58
3.15	Realisierung der Hauptplatine - Kompasssensor und Gumstix	70
3.16	Realisierung der Hauptplatine - Reset	71

3.17	Realisierung der Hauptplatine - Radencoder	71
3.18	Realisierung der Hauptplatine - Bürstenmotor	73
3.19	Realisierung der Hauptplatine - Radmotor	74
3.20	Realisierung der Hauptplatine - Akkumulator	75
3.21	Realisierung der Hauptplatine - Kontaktsensoren	76
3.22	Der optische Maussensor [7]	77
3.23	Bearbeitung der unteren Aluminiumplatte	78
3.24	Bearbeitung der unteren Aluminiumplatte	79
3.25	Bauteile zur Motormontage	79
3.26	Befestigung der Räder an der unteren Platte	80
3.27	Befestigung des Akkus an der unteren Platte	81
3.28	Bürsten	81
3.29	Befestigung der Bürsten an der unteren Platte	82
3.30	Befestigung der Stange an der unteren Platte	82
3.31	Ultraschallsensor an dem Metallstück	83
3.32	RAPTOR	83
4.1	Anschluss an das Carrier Board	92
4.2	Carrier Board mit Anschlusskabel	93
4.3	Bereichswachstumsverfahren [17]	99
4.4	Entfernungsschätzung für die Routenplanung	103
4.5	Die berechnete Route	103
4.6	erste Datenstruktur zur Verwaltung der Gitterkarte	107
4.7	finale Datenstruktur zur Verwaltung der Gitterkarte	107
4.8	Descartes' Zerlegung des Raumes [14]	109
4.9	Voronoi-Diagramm $V(S)$ [31]	110
4.10	(i) $x \notin V(S)$ (ii) $x \in V(S)$ und ist <i>Kante</i> (iii) $x \in V(S)$ und ist <i>Knoten</i> [14]	111
4.11	beschränktes Voronoi-Diagramm $V_0(S)$ [31]	112
4.12	Der Roboter befindet sich am Startpunkt s und findet ein nahe Voronoi- Kante [31]	113
4.13	Kein Punkt rechts der Sweep-Line L kann das Gebiet zur Linken von $B(p,$ $L)$ beeinflussen.	114
4.14	Drei Schritte der Berechnung eines Voronoi-Diagramms.	115
4.15	Komponentenuebersicht Player/Stage/Gazebo [21]	117
4.16	Simulationsumgebung	120
4.17	Simulationsumgebung	121
4.18	API	124
4.19	Anbindung von Geräten am I^2C -Bus [23]	131
4.20	SDA darf sich während eines SCK-Impulses nicht ändern [23]	131
4.21	Start- und Stopp-Bedingung [23]	132
4.22	Der Empfänger quittiert den Empfang eines Paketes, indem er die SDA-Ader beim "9. Bit" auf Null zieht. [23]	133

A.1	Untere Ebene von oben aus gesehen	139
A.2	Untere Ebene von unten aus gesehen	140
A.3	Obere Ebene von unten aus gesehen	141
A.4	Obere Ebene von oben aus gesehen	142
A.5	Testschaltung der Motoren	144
A.6	Schaltung der Maussensorplatine	145
A.7	Schaltplan der Hauptplatine	146
A.8	Schaltplan der Platine für den Lichtsensor	147
A.9	Schaltplan der Platine für die Bodenbeleuchtung	147
A.10	Schaltplan der Platine für die Leuchtdioden Betriebszustand und Akku- latorzustand	148
A.11	Testplatine der Motoren	149

Tabellenverzeichnis

2.1	Registerbelegung des Kompassmoduls	28
3.1	Pinbelegung der BreakoutBox [18]	43
3.2	Die Pinbelegung verschiedener Stecker	52
4.1	Beispiel für die Umrechnung von Koordinanten	105

Literaturverzeichnis

- [1] AGILENT-TECHNOLOGIES. <http://www.agilent.com>.
- [2] ATMEL. <http://www.atmel.com>.
- [3] BASCOM. <http://www.mcselec.com/bascom-avr.htm>.
- [4] BEISPIEL-APPLET. [http://www.personal.kent.edu/\\$\sim\\$rmuhamma/Compgeometry/MyCG/Voronoi/Fortune/fortune.htm](http://www.personal.kent.edu/$\sim$rmuhamma/Compgeometry/MyCG/Voronoi/Fortune/fortune.htm).
- [5] (BGL), BOOST GRAPH LIBRARY. <http://www.boost.org/libs/graph/doc/index.html>.
- [6] COMPUTING, MEASUREMENT. <http://www.measurementcomputing.com/>.
- [7] C'T-BOT. <http://www.heise.de/ct/projekte/ct-bot>.
- [8] DEVANTECH. <http://www.robot-electronics.co.uk>.
- [9] EAGLE. <http://www.cadsoft.de>.
- [10] GMBH, PLUG-IN ELECTRONICS. <http://www.plugin.de/>.
- [11] GUMSTIX. <http://www.gumstix.com>.
- [12] HARTMANN. <http://www.hartmann-codier.de>.
- [13] I2C, GUMSTIX WIKI. <http://www.davehylands.com/gumstix-wiki/i2c/>.
- [14] KLEIN, ROLF. Algorithmische Geometrie - Springer 2004 - ISBN: 978-3-540-20956-0.
- [15] KUNG-LONG. <http://www.klb.com.tw>.
- [16] LOGITECH. <http://www.logitech.com>.
- [17] MARSHALL, DAVID. http://www.cs.cf.ac.uk/Dave/Vision_lecture/node35.html.
- [18] MEASUREMENT-COMPUTING. <http://www.measurementcomputing.com/>.
- [19] O'SULLIVAN, SHANE. [http://www.skynet.ie/\\$\sim\\$ssos/mapviewer/voronoi.php](http://www.skynet.ie/$\sim$ssos/mapviewer/voronoi.php).

- [20] PATENTE-STUDIERENDE. <http://www.patente-studierende.de>.
- [21] PLAYER-STAGE-GAZEBO. <http://www-home.htwg-konstanz.de/~marcel/Robo/index.html>.
- [22] ROBOTERNETZ. http://www.roboternetz.de/wissen/index.php/AVR-ISP_Programmierkabel.
- [23] ROBOTERNETZ. <http://www.roboternetz.de/wissen>.
- [24] SGI. <http://www.sgi.com/tech/stl/>.
- [25] SGI. <http://www.sgi.com/tech/stl/Vector.html>.
- [26] VISHAY. <http://www.vishay.com>.
- [27] VX, GUMSTIX CONSOLE. <http://pubs.gumstix.org/boards/CONSOLE/VX/PCB10003-R1753>.
- [28] WIKI, GUMSTIX. <http://docwiki.gumstix.org>.
- [29] WIKIPEDIA. <http://de.wikipedia.org/wiki/Schwellwertverfahren>.
- [30] WIKIPEDIA. <http://de.wikipedia.org/wiki/Vektor>.
- [31] WIKIPEDIA. <http://de.wikipedia.org/wiki/Voronoi-Diagramm>.
- [32] WINAVR. <http://winavr.sourceforge.net/>.